Neural Network Toolbox

For Use with MATLAB®

Howard Demuth
Mark Beale

Computation

Visualization

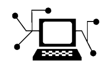
Programming



Version 4



How to Contact The MathWorks:



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup



`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information



508-647-7000 Phone



508-647-7001 Fax



The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Neural Network Toolbox User's Guide

Printing History:	June 1992	First printing	
	April 1993	Second printing	
	January 1997	Third printing	
	July 1997	Fourth printing	
	January 1998	Fifth printing	Revised for Version 3 (Release 11)
	September 2000	Sixth printing	Revised for Version 4 (Release 12)
	June 2001	Seventh printing	Minor revisions (Release 12.1)
	July 2002	Online only	Minor revisions (Release 13)

Contents

Preface

Neural Networks . . . . **xviii**

Basic Chapters . . . . **xx**

Mathematical Notation for Equations and Figures . . . . **xxi**
- Basic Concepts . . . . **xxi**
- Language . . . . **xxi**
- Weight Matrices . . . . **xxi**
- Layer Notation . . . . **xxi**
- Figure and Equation Examples . . . . **xxii**

Mathematics and Code Equivalents . . . . **xxiii**

Neural Network Design Book . . . . **xxiv**

Acknowledgments . . . . **xxv**

1 Introduction

Getting Started **1-2**
- Basic Chapters **1-2**
- Help and Installation **1-2**

What's New in Version 4.0 . . . . **1-3**
- Control System Applications **1-3**
- Graphical User Interface **1-3**
- New Training Functions **1-3**
- Design of General Linear Networks **1-4**
- Improved Early Stopping **1-4**

Generalization and Speed Benchmarks 1-4
Demonstration of a Sample Training Session 1-4

Neural Network Applications 1-5
Applications in this Toolbox 1-5
Business Applications 1-5
Aerospace 1-5
Automotive 1-5
Banking 1-5
Credit Card Activity Checking 1-5
Defense 1-6
Electronics 1-6
Entertainment 1-6
Financial 1-6
Industrial 1-6
Insurance 1-6
Manufacturing 1-6
Medical 1-7
Oil and Gas 1-7
Robotics 1-7
Speech 1-7
Securities 1-7
Telecommunications 1-7
Transportation 1-7
Summary 1-7

2 Neuron Model and Network Architectures

Neuron Model 2-2
Simple Neuron 2-2
Transfer Functions 2-3
Neuron with Vector Input 2-5

Network Architectures 2-8
A Layer of Neurons 2-8
Multiple Layers of Neurons 2-11

Data Structures 2-13
Simulation With Concurrent Inputs in a Static Network 2-13
Simulation With Sequential Inputs in a Dynamic Network . . 2-14
Simulation With Concurrent Inputs in a Dynamic Network . 2-16

Training Styles 2-18
Incremental Training (of Adaptive and Other Networks) 2-18
Batch Training 2-20

Summary 2-24
Figures and Equations 2-25

3 Perceptrons

Introduction 3-2
Important Perceptron Functions 3-3

Neuron Model 3-4

Perceptron Architecture 3-6

Creating a Perceptron (newp) 3-7
Simulation (sim) 3-8
Initialization (init) 3-9

Learning Rules 3-12

Perceptron Learning Rule (learnp) 3-13

Training (train) 3-16

Limitations and Cautions 3-21
Outliers and the Normalized Perceptron Rule 3-21

Graphical User Interface **3-23**
Introduction to the GUI **3-23**
Create a Perceptron Network (nntool) **3-23**
Train the Perceptron **3-27**
Export Perceptron Results to Workspace **3-29**
Clear Network/Data Window **3-30**
Importing from the Command Line **3-30**
Save a Variable to a File and Load It Later **3-31**

Summary **3-33**
Figures and Equations **3-33**
New Functions **3-36**

4 Linear Filters

Introduction **4-2**

Neuron Model **4-3**

Network Architecture **4-4**
Creating a Linear Neuron (newlin) **4-4**

Mean Square Error **4-8**

Linear System Design (newlind) **4-9**

Linear Networks with Delays **4-10**
Tapped Delay Line **4-10**
Linear Filter **4-10**

LMS Algorithm (learnwh) **4-13**

Linear Classification (train) **4-15**

Limitations and Cautions **4-18**
Overdetermined Systems **4-18**
Underdetermined Systems **4-18**
Linearly Dependent Vectors **4-18**
Too Large a Learning Rate **4-19**

Summary **4-20**
Figures and Equations **4-21**
New Functions **4-25**

5 Backpropagation

Overview **5-2**

Fundamentals **5-4**
Architecture **5-4**
Simulation (sim) **5-8**
Training **5-8**

Faster Training **5-14**
Variable Learning Rate (traingda, traingdx) **5-14**
Resilient Backpropagation (trainrp) **5-16**
Conjugate Gradient Algorithms **5-17**
Line Search Routines **5-23**
Quasi-Newton Algorithms **5-26**
Levenberg-Marquardt (trainlm) **5-28**
Reduced Memory Levenberg-Marquardt (trainlm) **5-30**

Speed and Memory Comparison **5-32**
Summary **5-49**

Improving Generalization **5-51**
Regularization **5-52**
Early Stopping **5-55**
Summary and Discussion **5-57**

Preprocessing and Postprocessing **5-61**
Min and Max (premnmx, postmnmx, tramnmx) **5-61**
Mean and Stand. Dev. (prestd, poststd, trastd) **5-62**
Principal Component Analysis (prepca, trapca) **5-63**
Post-Training Analysis (postreg) **5-64**

Sample Training Session **5-66**

Limitations and Cautions **5-71**

Summary **5-73**

6 Control Systems

Introduction **6-2**

NN Predictive Control **6-4**
System Identification **6-4**
Predictive Control **6-5**
Using the NN Predictive Controller Block **6-6**

NARMA-L2 (Feedback Linearization) Control **6-14**
Identification of the NARMA-L2 Model **6-14**
NARMA-L2 Controller **6-16**
Using the NARMA-L2 Controller Block **6-18**

Model Reference Control **6-23**
Using the Model Reference Controller Block **6-25**

Importing and Exporting **6-31**
Importing and Exporting Networks **6-31**
Importing and Exporting Training Data **6-35**

Summary **6-38**

7 Radial Basis Networks

Introduction 7-2
Important Radial Basis Functions 7-2

Radial Basis Functions 7-3
Neuron Model 7-3
Network Architecture 7-4
Exact Design (newrbe) 7-5
More Efficient Design (newrb) 7-7
Demonstrations 7-8

Generalized Regression Networks 7-9
Network Architecture 7-9
Design (newgrnn) 7-10

Probabilistic Neural Networks 7-12
Network Architecture 7-12
Design (newpnn) 7-13

Summary 7-15
Figures 7-16
New Functions 7-18

8 Self-Organizing and Learn. Vector Quant. Nets

Introduction 8-2
Important Self-Organizing and LVQ Functions 8-2

Competitive Learning 8-3
Architecture 8-3
Creating a Competitive Neural Network (newc) 8-4
Kohonen Learning Rule (learnk) 8-5
Bias Learning Rule (learncon) 8-5

Training 8-6
Graphical Example 8-7

Self-Organizing Maps 8-9
Topologies (gridtop, hextop, randtop) 8-10
Distance Funct. (dist, linkdist, mandist, boxdist) 8-14
Architecture 8-17
Creating a Self Organizing MAP Neural Network (newsom) . 8-18
Training (learnsom) 8-19
Examples 8-23

Learning Vector Quantization Networks 8-31
Architecture 8-31
Creating an LVQ Network (newlvq) 8-32
LVQ1 Learning Rule(learnlv1) 8-35
Training 8-36
Supplemental LVQ2.1 Learning Rule (learnlv2) 8-38

Summary and Conclusions 8-40
Self-Organizing Maps 8-40
Learning Vector Quantizaton Networks 8-40
Figures 8-41
New Functions 8-42

9 Recurrent Networks

Introduction 9-2
Important Recurrent Network Functions 9-2

Elman Networks 9-3
Architecture 9-3
Creating an Elman Network (newelm) 9-4
Training an Elman Network 9-5

Hopfield Network 9-8
Fundamentals 9-8
Architecture 9-8
Design (newhop) 9-10

Summary 9-15
Figures 9-16
New Functions 9-17

10 Adaptive Filters and Adaptive Training

Introduction 10-2
Important Adaptive Functions 10-2

Linear Neuron Model 10-3

Adaptive Linear Network Architecture 10-4
Single ADALINE (newlin) 10-4

Mean Square Error 10-7

LMS Algorithm (learnwh) 10-8

Adaptive Filtering (adapt) 10-9
Tapped Delay Line 10-9
Adaptive Filter 10-9
Adaptive Filter Example 10-10
Prediction Example 10-13
Noise Cancellation Example 10-14
Multiple Neuron Adaptive Filters 10-16

Summary 10-18
Figures and Equations 10-18
New Functions 10-26

11 Applications

Introduction **11-2**
Application Scripts **11-2**

Applin1: Linear Design **11-3**
Problem Definition **11-3**
Network Design **11-4**
Network Testing **11-4**
Thoughts and Conclusions **11-6**

Applin2: Adaptive Prediction **11-7**
Problem Definition **11-7**
Network Initialization **11-8**
Network Training **11-8**
Network Testing **11-8**
Thoughts and Conclusions **11-10**

Appelm1: Amplitude Detection **11-11**
Problem Definition **11-11**
Network Initialization **11-11**
Network Training **11-12**
Network Testing **11-13**
Network Generalization **11-13**
Improving Performance **11-15**

Appcr1: Character Recognition **11-16**
Problem Statement **11-16**
Neural Network **11-17**
System Performance **11-20**
Summary **11-22**

12 Advanced Topics

Custom Networks 12-2
Custom Network 12-3
Network Definition 12-4
Network Behavior 12-12

Additional Toolbox Functions 12-16
Initialization Functions 12-16
Transfer Functions 12-16
Learning Functions 12-17

Custom Functions 12-18
Simulation Functions 12-18
Initialization Functions 12-24
Learning Functions 12-27
Self-Organizing Map Functions 12-36

13 Network Object Reference

Network Properties 13-2
Architecture 13-2
Subobject Structures 13-6
Functions 13-9
Parameters 13-12
Weight and Bias Values 13-14
Other 13-16

Subobject Properties 13-17
Inputs 13-17
Layers 13-18
Outputs 13-25
Targets 13-25
Biases 13-26
Input Weights 13-28
Layer Weights 13-32

14 Reference

Functions — By Category **14-2**
Functions by Network Type **14-2**
Functions by Class **14-2**
Transfer Functions **14-13**
Transfer Function Graphs **14-14**

Functions — Alphabetical List **14-19**
Reference Headings **14-19**

A Glossary

B Bibliography

C Demonstrations and Applications

Tables of Demonstrations and Applications **C-2**
Chapter 2: Neuron Model and Network Architectures **C-2**
Chapter 3: Perceptrons **C-2**
Chapter 4: Linear Filters **C-3**
Chapter 5: Backpropagation **C-3**
Chapter 7: Radial Basis Networks **C-4**
Chapter 8: Self-Organizing and Learn. Vector Quant. Nets . . . **C-4**
Chapter 9: Recurrent Networks **C-4**
Chapter 10: Adaptive Networks **C-5**
Chapter 11: Applications **C-5**

D Simulink

Block Set **D-2**
Transfer Function Blocks **D-2**
Net Input Blocks **D-3**
Weight Blocks **D-3**
Control Systems Blocks **D-4**

Block Generation **D-5**
Example **D-5**
Exercises **D-7**

E Code Notes

Dimensions **E-2**

Variables **E-3**
Utility Function Variables **E-4**

Functions **E-7**

Code Efficiency **E-8**

Argument Checking **E-9**

Index

Preface

Neural Networks xviii

Basic Chapters xx

Mathematical Notation for Equations and Figures . . xxi
Basic Concepts xxi
Language xxi
Weight Matrices xxi
Layer Notation xxi
Figure and Equation Examples xxii

Mathematics and Code Equivalents xxiii

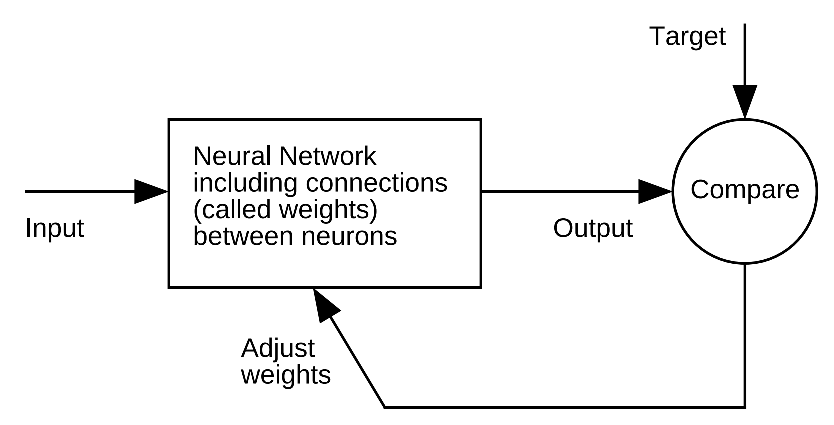
Neural Network Design Book xxiv

Acknowledgments xxv

Neural Networks

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learning*, to train a network.



Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as “on line” or “adaptive” training.

Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1.

Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

The supervised training methods are commonly used, but other networks can be obtained from *unsupervised training* techniques or from direct *design* methods. Unsupervised networks can be used, for instance, to identify groups of data. Certain kinds of linear networks and Hopfield networks are designed directly. In summary, there are a variety of kinds of design and learning techniques that enrich the choices that a user can make.

The field of neural networks has a history of some five decades but has found solid application only in the past fifteen years, and the field is still developing rapidly. Thus, it is distinctly different from the fields of control systems or optimization where the terminology, basic mathematics, and design procedures have been firmly established and applied for many years. We do not view the Neural Network Toolbox as simply a summary of established procedures that are known to work well. Rather, we hope that it will be a useful tool for industry, education and research, a tool that will help users find what works and what doesn't, and a tool that will help develop and extend the field of neural networks. Because the field and the material are so new, this toolbox will explain the procedures, tell how to apply them, and illustrate their successes and failures with examples. We believe that an understanding of the paradigms and their application is essential to the satisfactory and successful use of this toolbox, and that without such understanding user complaints and inquiries would bury us. So please be patient if we include a lot of explanatory material. We hope that such material will be helpful to you.

Basic Chapters

The Neural Network Toolbox is written so that if you read Chapter 2, Chapter 3 and Chapter 4 you can proceed to a later chapter, read it and use its functions without difficulty. To make this possible, Chapter 2 presents the fundamentals of the neuron model, the architectures of neural networks. It also will discuss notation used in the architectures. All of this is basic material. It is to your advantage to understand this Chapter 2 material thoroughly.

The neuron model and the architecture of a neural network describe how a network transforms its input into an output. This transformation can be viewed as a computation. The model and the architecture each place limitations on what a particular neural network can compute. The way a network computes its output must be understood before training methods for the network can be explained.

Mathematical Notation for Equations and Figures

Basic Concepts

Scalars-small *italic* letters.....*a,b,c*

Vectors - small **bold** non-italic letters.....**a,b,c**

Matrices - capital **BOLD** non-italic letters.....**A,B,C**

Language

Vector means a column of numbers.

Weight Matrices

Scalar Element $w_{i,j}(t)$

i - row, j - column, t - time or iteration

Matrix $\mathbf{W}(t)$

Column Vector $\mathbf{w}_j(t)$

Row Vector ${}_i\mathbf{w}(t)$ **...**vector made of ith row of weight matrix **W**

Bias Vector

Scalar Element $b_i(t)$

Vector $\mathbf{b}(t)$

Layer Notation

A single superscript is used to identify elements of layer. For instance, the net input of layer 3 would be shown as $\mathbf{n}^3$.

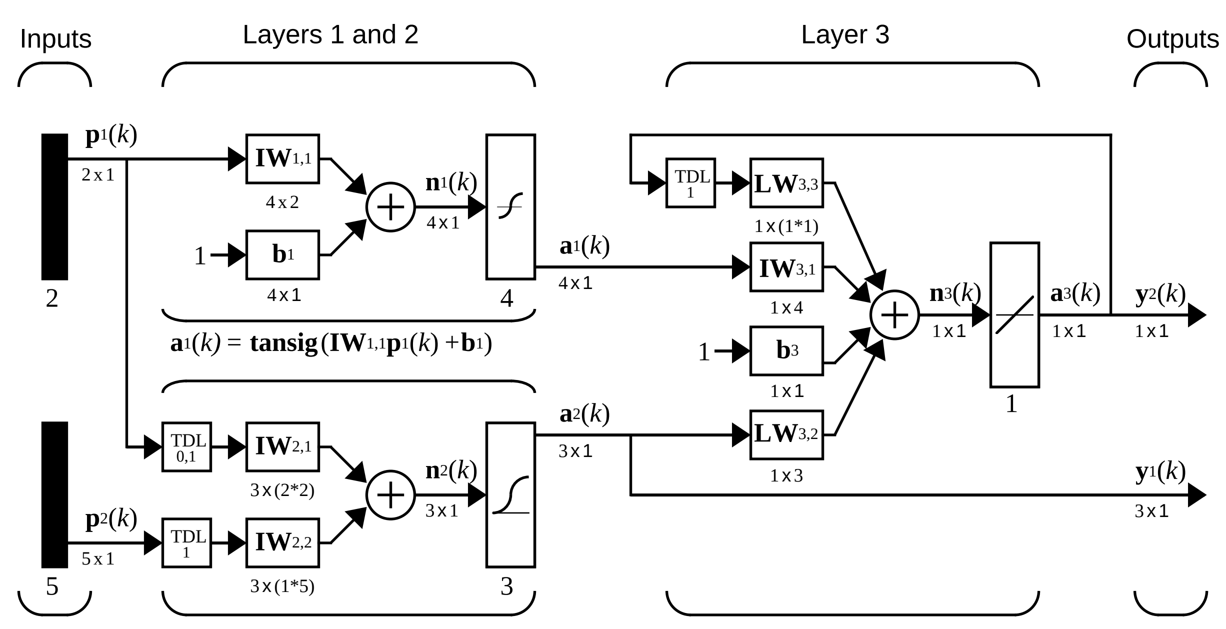
Superscripts k, l are used to identify the source (l) connection and the destination (k) connection of layer weight matrices ans input weight matrices. For instance, the layer weight matrix from layer 2 to layer 4 would be shown as $\mathbf{LW}^{4,2}$.

Input Weight Matrix $\mathbf{IW}^{k,l}$

Layer Weight Matrix $\mathbf{LW}^{k,l}$

Figure and Equation Examples

The following figure, taken from Chapter 12 illustrates notation used in such advanced figures.



Mathematics and Code Equivalents

The transition from mathematics to code or vice versa can be made with the aid of a few rules. They are listed here for future reference.

To change from mathematics notation to MATLAB notation, the user needs to:

- Change superscripts to cell array indices.
  For example, $p^{1} \rightarrow p\{1\}$
- Change subscripts to parentheses indices.
  For example, $p_{2} \rightarrow p(2)$, and $p_{2}^{1} \rightarrow p\{1\}(2)$
- Change parentheses indices to a second cell array index.
  For example, $p^{1}(k-1) \rightarrow p\{1, k-1\}$
- Change mathematics operators to MATLAB operators and toolbox functions.
  For example, $ab \rightarrow a$*b

The following equations illustrate the notation used in figures.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Neural Network Design Book

Professor Martin Hagan of Oklahoma State University, and Neural Network Toolbox authors Howard Demuth and Mark Beale have written a textbook, *Neural Network Design*, published by the Brooks/Cole Publishing Company in 1996 (ISBN 0-534-94332-2). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of MATLAB and the Neural Network Toolbox. Demonstration programs from the book are used in various chapters of this Guide. (You can find all the book demonstration programs in the Neural Network Toolbox by typing `nnd`.)

The book has:

- An INSTRUCTOR'S MANUAL (ISBN 0-534-95049-3) for adopters and
- TRANSPARENCY OVERHEADS for class use. The overheads come one to a page for instructor use and three to a page for student use.

To place an order for the book, call 1-800-354-9706.

To obtain a copy of the INSTRUCTOR'S MANUAL, contact the Brooks/Cole Editorial Office, phone 1-800-354-0092. Ask specifically for an instructor's manual if you are instructing a class and want one.

You can go directly to the Brooks/Cole Neural Network Design page at

```
http://brookscole.com/engineering/nnd.html
```

Once there, you can download the TRANSPARENCY MASTERS with a click on "Transparency Masters(3.6MB)."

Alternatively, you might try the Brooks/Cole Thomson Leaning Web site home page:

```
http://brookscole.com
```

One there, pick Engineering, then Electrical Engineering and finally "Browse." This will get you a list of books, including Neural Networks Design.

Acknowledgments

The authors would like to thank:

Martin Hagan of Oklahoma State University for providing the original Levenberg-Marquardt algorithm in the Neural Network Toolbox version 2.0 and various algorithms found in version 3.0, including the new reduced memory use version of the Levenberg-Marquardt algorithm, the coujugate gradient algorithm, RPROP, and generalized regression method. Martin also wrote Chapter 5 and Chapter 6 of this toolbox. Chapter 5 describes new algorithms, suggests algorithms for pre- and post-processing of data, and presents a comparison of the efficacy of various algorithms. Chapter 6 on control system applications, describes practical applications including neural network model predictive control, model reference adaptive control, and a feedback linearization controller.

Joe Hicklin of The MathWorks for getting Howard into neural network research years ago at the University of Idaho, for encouraging Howard to write the toolbox, for providing crucial help in getting the first toolbox version 1.0 out the door, and for continuing to be a good friend.

Jim Tung of The MathWorks for his long-term support for this project.

Liz Callanan of The MathWorks for getting us off the such a good start with the Neural Network Toolbox version 1.0.

Roy Lurie of The MathWorks for his vigilant reviews of the developing material in this version of the toolbox.

Matthew Simoneau of The MathWorks for his help with demos, test suite routines, for getting user feedback, and for helping with other toolbox matters.

Sean McCarthy for his many questions from users about the toolbox operation

Jane Carmody of The MathWorks for editing help and for always being at her phone to help with documentation problems.

Donna Sullivan and **Peg Theriault** of The MathWorks for their editing and other help with the Mac document.

Jane Price of The MathWorks for getting constructive user feedback on the toolbox document and its graphical user interface.

Orlando De Jesús of Oklahoma State University for his excellent work in programming the neural network controllers described in Chapter 6.

Bernice Hewitt for her wise New Zealand counsel, encouragement, and tea, and for the company of her cats **Tiny** and **Mr. Britches**.

Joan Pilgram for her business help, general support, and good cheer.

Teri Beale for running the show and having Valerie and Asia Danielle while Mark worked on this toolbox.

Martin Hagan and Howard Demuth for permission to include various problems, demonstrations, and other material from *Neural Network Design*, Jan. 1996.

1

Introduction

Getting Started . . . 1-2
Basic Chapters . . . 1-2
Help and Installation . . . 1-2

What's New in Version 4.0 . . . 1-3
Control System Applications . . . 1-3
Graphical User Interface . . . 1-3
New Training Functions . . . 1-3
Design of General Linear Networks . . . 1-4
Improved Early Stopping . . . 1-4
Generalization and Speed Benchmarks . . . 1-4
Demonstration of a Sample Training Session . . . 1-4

Neural Network Applications . . . 1-5
Applications in this Toolbox . . . 1-5
Business Applications . . . 1-5
Aerospace . . . 1-5
Automotive . . . 1-5
Banking . . . 1-5
Credit Card Activity Checking . . . 1-5
Defense . . . 1-6
Electronics . . . 1-6
Entertainment . . . 1-6
Financial . . . 1-6
Industrial . . . 1-6
Insurance . . . 1-6
Manufacturing . . . 1-6
Medical . . . 1-7
Oil and Gas . . . 1-7
Robotics . . . 1-7
Speech . . . 1-7
Securities . . . 1-7
Telecommunications . . . 1-7
Transportation . . . 1-7
Summary . . . 1-7

Getting Started

Basic Chapters

Chapter 2 contains basic material about network architectures and notation specific to this toolbox.Chapter 3 includes the first reference to basic functions such as `init` and `adapt`. Chapter 4 describes the use of the functions `designd` and `train`, and discusses delays. Chapter 2, Chapter 3, and Chapter 4 should be read before going to later chapters

Help and Installation

The Neural Network Toolbox is contained in a directory called `nnet`. Type `help nnet` for a listing of help topics.

A number of demonstrations are included in the toolbox. Each example states a problem, shows the network used to solve the problem, and presents the final results. Lists of the neural network demonstration and application scripts that are discussed in this guide can be found by typing `help nndemos`

Instructions for installing the Neural Network Toolbox are found in one of two MATLAB documents: the *Installation Guide for PC* or the *Installation Guide for UNIX*.

What's New in Version 4.0

A few of the new features and improvements introduced with this version of the Neural Network Toolbox are discussed below.

Control System Applications

A new Chapter 6 presents three practical control systems applications:

- Network model predictive control
- Model reference adaptive control
- Feedback linearization controller

Graphical User Interface

A graphical user interface has been added to the toolbox. This interface allows you to:

- Create networks
- Enter data into the GUI
- Initialize, train, and simulate networks
- Export the training results from the GUI to the command line workspace
- Import data from the command line workspace to the GUI

To open the Network/Data Manager window type `nntool`.

New Training Functions

The toolbox now has four training algorithms that apply weight and bias learning rules. One algorithm applies the learning rules in batch mode. Three algorithms apply learning rules in three different incremental modes:

- `trainb` - Batch training function
- `trainc` - Cyclical order incremental training function
- `trainr` - Random order incremental training function
- `trains` - Sequential order incremental training function

All four functions present the whole training set in each epoch (pass through the entire input set).

Note We no longer recommend using `trainwb` and `trainwb1`, which have been replaced by `trainb` and `trainr`. The function `trainr` differs from `trainwb1` in that `trainwb1` only presented a single vector each epoch instead of going through all vectors, as is done by `trainr`.

These new training functions are relatively fast because they generate M-code. The functions `trainb`, `trainc`, `trainr`, and `trains` all generate a temporary M-file consisting of specialized code for training the current network in question.

Design of General Linear Networks

The function `newlind` now allows the design of linear networks with multiple inputs, outputs, and input delays.

Improved Early Stopping

Early stopping can now be used in combination with Bayesian regularization. In some cases this can improve the generalization capability of the trained network.

Generalization and Speed Benchmarks

Generalization benchmarks comparing the performance of Bayesian regularization and early stopping are provided. We also include speed benchmarks, which compare the speed of convergence of the various training algorithms on a variety of problems in pattern recognition and function approximation. These benchmarks can aid users in selecting the appropriate algorithm for their problem.

Demonstration of a Sample Training Session

A new demonstration that illustrates a sample training session is included in Chapter 5. A sample training session script is also provided. Users can modify this script to fit their problem.

Neural Network Applications

Applications in this Toolbox

Chapter 6 describes three practical neural network control system applications, including neural network model predictive control, model reference adaptive control, and a feedback linearization controller.

Other neural network applications are described in Chapter 11.

Business Applications

The *1988 DARPA Neural Network Study* [DARP88] lists various neural network applications, beginning in about 1984 with the adaptive channel equalizer. This device, which is an outstanding commercial success, is a single-neuron network used in long-distance telephone systems to stabilize voice signals. The *DARPA* report goes on to list other commercial applications, including a small word recognizer, a process monitor, a sonar classifier, and a risk analysis system.

Neural networks have been applied in many other fields since the *DARPA* report was written. A list of some applications mentioned in the literature follows.

Aerospace

- High performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, aircraft component fault detection

Automotive

- Automobile automatic guidance system, warranty activity analysis

Banking

- Check and other document reading, credit application evaluation

Credit Card Activity Checking

- Neural networks are used to spot unusual credit card activity that might possibly be associated with loss of a credit card

Defense

- Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, signal/image identification

Electronics

- Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, nonlinear modeling

Entertainment

- Animation, special effects, market forecasting

Financial

- Real estate appraisal, loan advisor, mortgage screening, corporate bond rating, credit-line use analysis, portfolio trading program, corporate financial analysis, currency price prediction

Industrial

- Neural networks are being trained to predict the output gasses of furnaces and other industrial processes. They then replace complex and costly equipment used for this purpose in the past.

Insurance

- Policy application evaluation, product optimization

Manufacturing

- Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, dynamic modeling of chemical process system

Medical

- Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency-room test advisement

Oil and Gas

- Exploration

Robotics

- Trajectory control, forklift robot, manipulator controllers, vision systems

Speech

- Speech recognition, speech compression, vowel classification, text-to-speech synthesis

Securities

- Market analysis, automatic bond rating, stock trading advisory systems

Telecommunications

- Image and data compression, automated information services, real-time translation of spoken language, customer payment processing systems

Transportation

- Truck brake diagnosis systems, vehicle scheduling, routing systems

Summary

The list of additional neural network applications, the money that has been invested in neural network software and hardware, and the depth and breadth of interest in these devices have been growing rapidly. The authors hope that this toolbox will be useful for neural network educational and design purposes within a broad field of neural network applications.

2

Neuron Model and Network Architectures

Neuron Model 2-2
Simple Neuron 2-2
Transfer Functions 2-3
Neuron with Vector Input 2-5

Network Architectures 2-8
A Layer of Neurons 2-8
Multiple Layers of Neurons 2-11

Data Structures 2-14
Simulation With Concurrent Inputs in a Static Network . . 2-14
Simulation With Sequential Inputs in a Dynamic Network . 2-15
Simulation With Concurrent Inputs in a Dynamic Network . 2-17

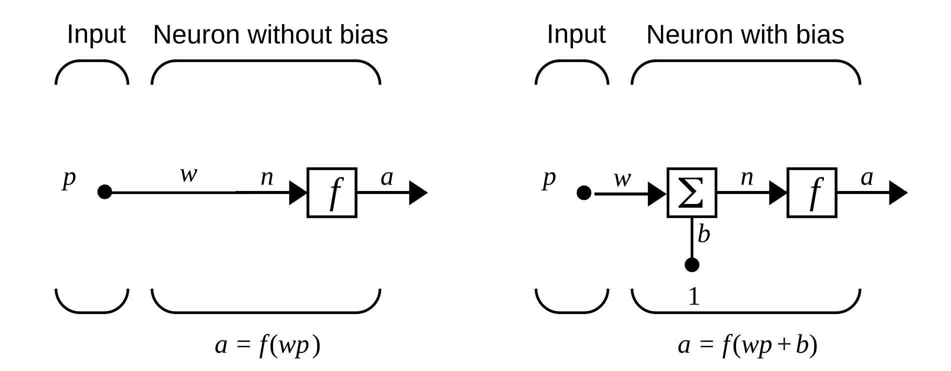
Training Styles 2-20
Incremental Training (of Adaptive and Other Networks) . . 2-20
Batch Training 2-22

Summary 2-27
Figures and Equations 2-28

Neuron Model

Simple Neuron

A neuron with a single scalar input and no bias appears on the left below.



The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w, to form the product wp, again a scalar. Here the weighted input wp is the only argument of the transfer function f, which produces the scalar output a. The neuron on the right has a scalar bias, b. You may view the bias as simply being added to the product wp as shown by the summing junction or as shifting the function f to the left by an amount b. The bias is much like a weight, except that it has a constant input of 1.

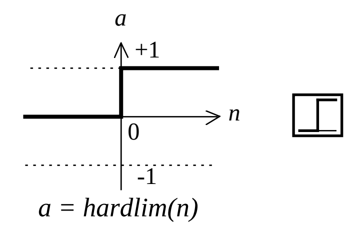
The transfer function net input n, again a scalar, is the sum of the weighted input wp and the bias b. This sum is the argument of the transfer function f. (Chapter 7 discusses a different way to form the net input n.) Here f is a transfer function, typically a step function or a sigmoid function, which takes the argument n and produces the output a. Examples of various transfer functions are given in the next section. Note that w and b are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, we can train the network to do a particular job by adjusting the weight or bias parameters, or perhaps the network itself will adjust these parameters to achieve some desired end.

All of the neurons in this toolbox have provision for a bias, and a bias is used in many of our examples and will be assumed in most of this toolbox. However, you may omit a bias in a neuron if you want.

As previously noted, the bias b is an adjustable (scalar) parameter of the neuron. It is *not* an input. However, the constant *1* that drives the bias is an input and must be treated as such when considering the linear dependence of input vectors in Chapter 4, "Linear Filters."

Transfer Functions

Many transfer functions are included in this toolbox. A complete list of them can be found in "Transfer Function Graphs" in Chapter 14. Three of the most commonly used functions are shown below.



Hard-Limit Transfer Function

The hard-limit transfer function shown above limits the output of the neuron to either 0, if the net input argument n is less than 0; or 1, if n is greater than or equal to 0. We will use this function in Chapter 3 "Perceptrons" to create neurons that make classification decisions.

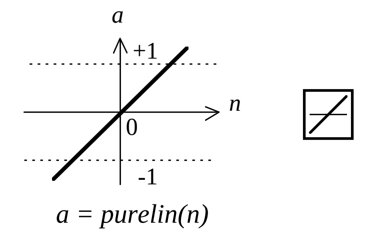
The toolbox has a function, `hardlim`, to realize the mathematical hard-limit transfer function shown above. Try the code shown below.

```
n = -5:0.1:5;
plot(n,hardlim(n),'c+:');
```

It produces a plot of the function `hardlim` over the range -5 to +5.

All of the mathematical transfer functions in the toolbox can be realized with a function having the same name.

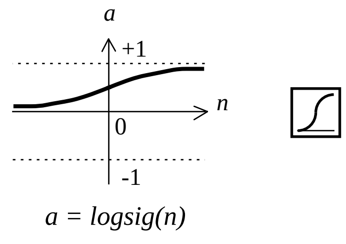
The linear transfer function is shown below.



Linear Transfer Function

Neurons of this type are used as linear approximators in “Linear Filters” in Chapter 4.

The sigmoid transfer function shown below takes the input, which may have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



Log-Sigmoid Transfer Function

This transfer function is commonly used in backpropagation networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons will replace the general *f* in the boxes of network diagrams to show the particular transfer function being used.

For a complete listing of transfer functions and their icons, see the “Transfer Function Graphs” in Chapter 14. You can also specify your own transfer functions. You are not limited to the transfer functions listed in Chapter 14.

You can experiment with a simple neuron and various transfer functions by running the demonstration program `nnd2n1`.

Neuron with Vector Input

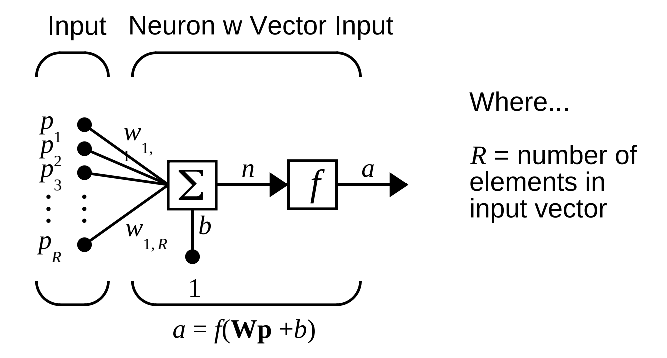
A neuron with a single R-element input vector is shown below. Here the individual element inputs

$$p_1, p_2, \ldots p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \ldots w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply **Wp**, the dot product of the (single row) matrix **W** and the vector **p**.



The neuron has a bias b, which is summed with the weighted inputs to form the net input n. This sum, n, is the argument of the transfer function f.

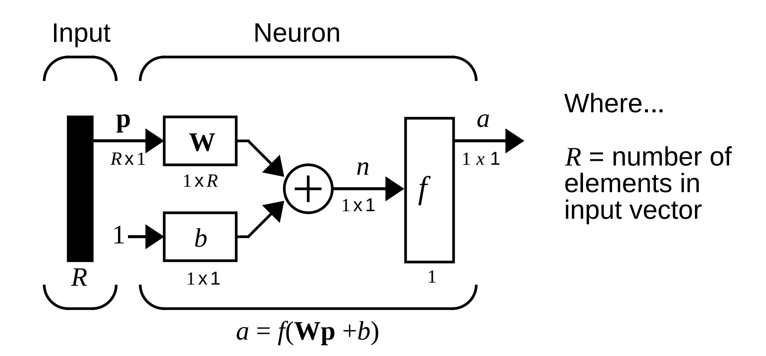
$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

This expression can, of course, be written in MATLAB code as:

```
n = W*p + b
```

However, the user will seldom be writing code at this low level, for such code is already built into functions to define and simulate entire networks.

The figure of a single neuron shown above contains a lot of detail. When we consider networks with many neurons and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which will be used later in circuits of multiple neurons, is illustrated in the diagram shown below.

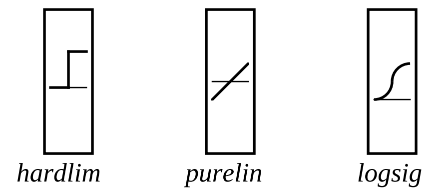


Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as Rx1. (Note that we will use a capital letter, such as R in the previous sentence, when referring to the *size* of a vector.) Thus, **p** is a vector of R input elements. These inputs post multiply the single row, R column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias b. The net input to the transfer function f is n, the sum of the bias b and the product **Wp**. This sum is passed to the transfer function f to get the neuron's output a, which in this case is a scalar. Note that if we had more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the figure shown above. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product **Wp**), the bias b, and the transfer function f. The array of inputs, vector **p**, is not included in or called a layer.

Each time this abbreviated network notation is used, the size of the matrices will be shown just below their matrix variable names. We hope that this notation will allow you to understand the architectures and follow the matrix mathematics associated with them.

As discussed previously, when a specific transfer function is to be used in a figure, the symbol for that transfer function will replace the *f* shown above. Here are some examples.



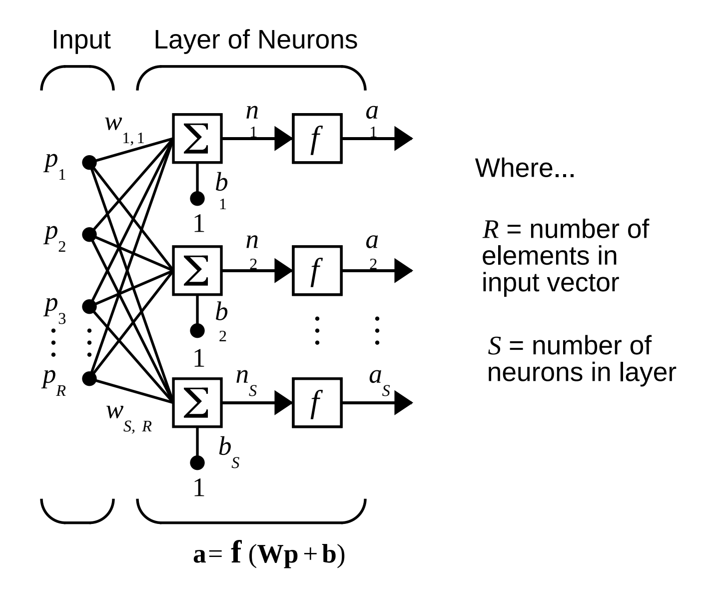
You can experiment with a two-element neuron by running the demonstration program `nnd2n2`.

Network Architectures

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

A Layer of Neurons

A one-layer network with R input elements and S neurons follows.



In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$. The various $n(i)$ taken together form an S-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. We show the expression for **a** at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., *R* ¦ *S*). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

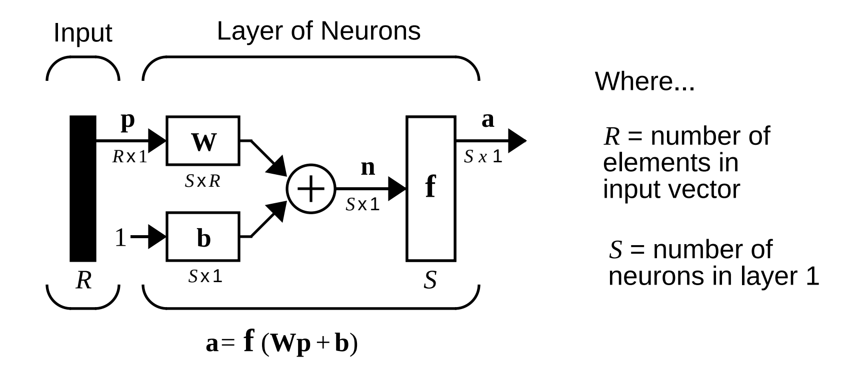
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R input one-layer network also can be drawn in abbreviated notation.

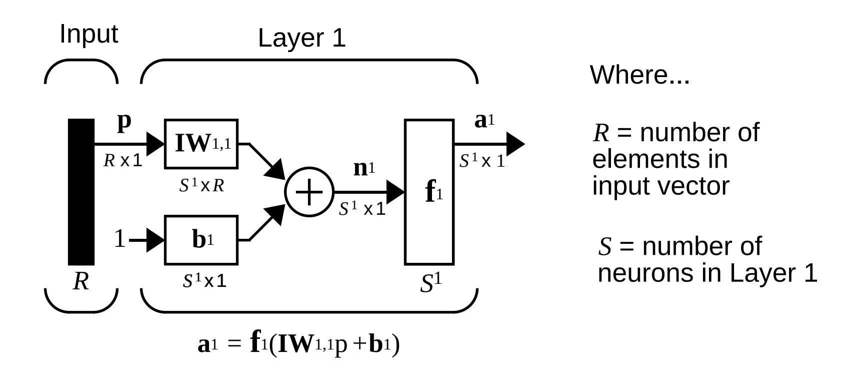


Here **p** is an R length input vector, **W** is an SxR matrix, and **a** and **b** are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function boxes.

Inputs and Layers

We are about to discuss networks having multiple layers so we will need to extend our notation to talk about such networks. Specifically, we need to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. We also need to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs, *input weights;* and we will call weight matrices coming from layer outputs, *layer weights.* Further, we will use superscripts to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, we have taken the one-layer multiple input network shown earlier and redrawn it in abbreviated form below.



As you can see, we have labeled the weight matrix connected to the input vector **p** as an Input Weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Also, elements of layer one, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

In the next section, we will use Layer Weight (**LW**) matrices as well as Input Weight (**IW**) matrices.

You might recall from the notation section of the Preface that conversion of the layer weight matrix from math to code for a particular network called *net* is:

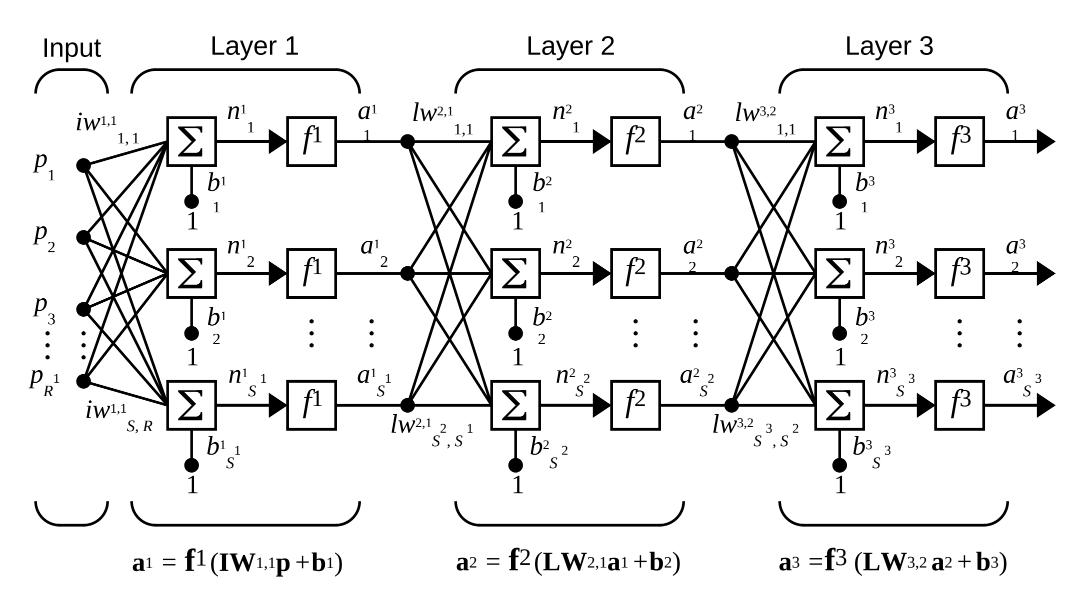
$$\mathbf{IW}^{1,\,1} \rightarrow \text{net.IW}\{1,\,1\}$$

Thus, we could write the code to obtain the net input to the transfer function as:

```
n{1} = net.IW{1,1}*p + net.b{1};
```

Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in our figures, we append the number of the layer as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown below, and in the equations at the bottom of the figure.



$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathrm{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

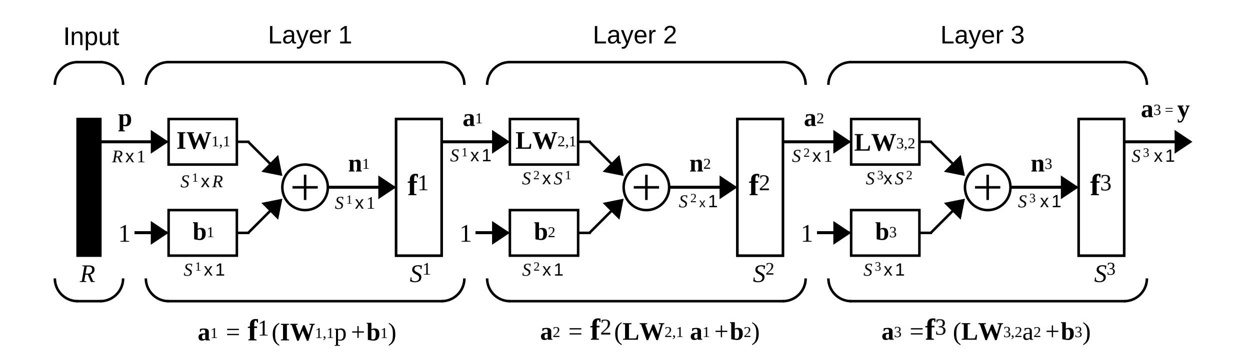
The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the biases for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an S^2xS^1 weight matrix W^2. The input to layer 2 is $\mathbf{a}^1$; the output

is $\mathbf{a}^2$. Now that we have identified all the vectors and matrices of layer 2, we can treat it as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. We will not use that designation.

The same three-layer network discussed previously also can be drawn using our abbreviated notation.



$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3 = \mathbf{y}$$

Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in Chapter 5, “Backpropagation.”

Here we assume that the output of the third layer, $\mathbf{a}^3$, is the network output of interest, and we have labeled this output as $\mathbf{y}$. We will use this notation to specify the output of multilayer networks.

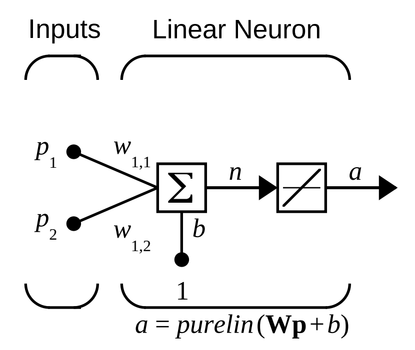
Data Structures

This section discusses how the format of input data structures affects the simulation of networks. We will begin with static networks, and then move to dynamic networks.

We are concerned with two basic types of input vectors: those that occur *concurrently* (at the same time, or in no particular time sequence), and those that occur *sequentially* in time. For concurrent vectors, the order is not important, and if we had a number of networks running in parallel, we could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

Simulation With Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, we do not have to be concerned about whether or not the input vectors occur in a particular time sequence, so we can treat the inputs as concurrent. In addition, we make the problem even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this feedforward network, we can use the following command.

```
net = newlin([1 3;1 3],1);
```

For simplicity assign the weight matrix and bias to be

$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \end{bmatrix}$.

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

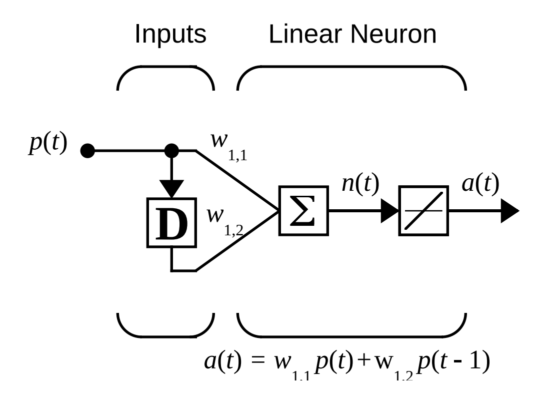
We can now simulate the network:

```
A = sim(net,P)
A =
     5     4     8     5
```

A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.

Simulation With Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.



The following commands create this network:

```
net = newlin([-1 1],1,[0 1]);
net.biasConnect = 0;
```

Assign the weight matrix to be

$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$.

The command is

```
net.IW{1,1} = [1 2];
```

Suppose that the input sequence is

$$\mathbf{p}1 = \begin{bmatrix} 1 \end{bmatrix},\ \mathbf{p}2 = \begin{bmatrix} 2 \end{bmatrix},\ \mathbf{p}3 = \begin{bmatrix} 3 \end{bmatrix},\ \mathbf{p}4 = \begin{bmatrix} 4 \end{bmatrix},$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

We can now simulate the network:

```
A = sim(net,P)
A =
    [1]    [4]    [7]    [10]
```

We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case,

the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If we were to change the order of the inputs, it would change the numbers we would obtain in the output.

Simulation With Concurrent Inputs in a Dynamic Network

If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. (Although, it is not clear why we would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if we use a concurrent set of inputs we have

$$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix},\quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix},\quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix},\quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When we simulate with concurrent inputs we obtain

```
A = sim(net,P)
A =
     1     2     3     4
```

The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.

In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let’s say we wanted to present the following two sequences to the network:

$$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix},\ \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix},\ \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

```
P = {[1 4] [2 3] [3 2] [4 1]};
```

We can now simulate the network:

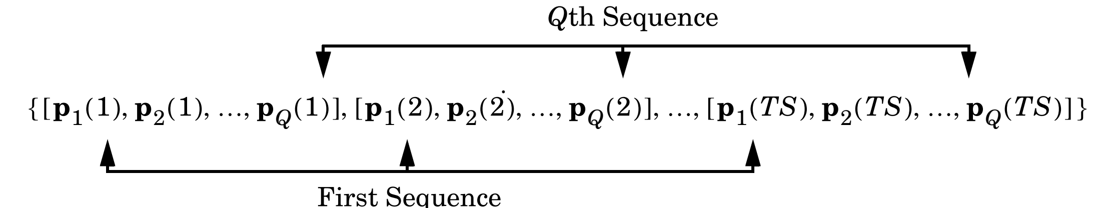
```
A = sim(net,P);
```

The resulting network output would be

```
A = {[ 1 4] [4 11] [7 8] [10 5]}
```

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the input P to the sim function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

Training Styles

In this section, we describe two different styles of training. In *incremental* training the weights and biases of the network are updated each time an input is presented to the network. In *batch* training the weights and biases are only updated after all of the inputs are presented.

Incremental Training (of Adaptive and Other Networks)

Incremental training can be applied to both static and dynamic networks, although it is more commonly used with dynamic networks, such as adaptive filters. In this section, we demonstrate how incremental training is performed on both static and dynamic networks.

Incremental Training with Static Networks

Consider again the static network we used for our first example. We want to train it incrementally, so that the weights and biases will be updated after each input is presented. In this case we use the function `adapt`, and we present the inputs and targets as sequences.

Suppose we want to train the network to create the linear function

$$t = 2p_1 + p_2 .$$

Then for the previous inputs we used,

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix},\ \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix},\ \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix},\ \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

the targets would be

$$\mathbf{t}_1 = \begin{bmatrix} 4 \end{bmatrix},\ \mathbf{t}_2 = \begin{bmatrix} 5 \end{bmatrix},\ \mathbf{t}_3 = \begin{bmatrix} 7 \end{bmatrix},\ \mathbf{t}_4 = \begin{bmatrix} 7 \end{bmatrix}$$

We first set up the network with zero initial weights and biases. We also set the learning rate to zero initially, to show the effect of the incremental training.

```
net = newlin([-1 1;-1 1],1,0,0);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For incremental training we want to present the inputs and targets as sequences:

```
P = {[1;2] [2;1] [2;3] [3;1]};
T = {4 5 7 7};
```

Recall from the earlier discussion that for a static network the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. This is not true when training the network, however. When using the `adapt` function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As we see in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

We are now ready to train the network incrementally.

```
[net,a,e,pf] = adapt(net,P,T);
```

The network outputs will remain zero, since the learning rate is zero, and the weights are not updated. The errors will be equal to the targets:

```
a = [0]    [0]    [0]    [0]
e = [4]    [5]    [7]    [7]
```

If we now set the learning rate to 0.1 we can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr=0.1;
net.biases{1,1}.learnParam.lr=0.1;
[net,a,e,pf] = adapt(net,P,T);
a = [0]    [2]    [6.0]    [5.8]
e = [4]    [3]    [1.0]    [1.2]
```

The first output is the same as it was with zero learning rate, since no update is made until the first input is presented. The second output is different, since the weights have been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error will eventually be driven to zero.

Incremental Training with Dynamic Networks

We can also train dynamic networks incrementally. In fact, this would be the most common situation. Let's take the linear network with one delay at the

input that we used in a previous example. We initialize the weights to zero and set the learning rate to 0.1.

```
net = newlin([-1 1],1,[0 1],0.1);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
```

To train this network incrementally we present the inputs and targets as elements of cell arrays.

```
Pi = {1};
P = {2 3 4};
T = {3 5 7};
```

Here we attempt to train the network to sum the current and previous inputs to create the current output. This is the same input sequence we used in the previous example of using `sim`, except that we assign the first term in the sequence as the initial condition for the delay. We now can sequentially train the network using adapt.

```
[net,a,e,pf] = adapt(net,P,T,Pi);
a = [0] [2.4] [ 7.98]
e = [3] [2.6] [-0.98]
```

The first output is zero, since the weights have not yet been updated. The weights change at each subsequent time step.

Batch Training

Batch training, in which weights and biases are only updated after all of the inputs and targets are presented, can be applied to both static and dynamic networks. We discuss both types of networks in this section.

Batch Training with Static Networks

Batch training can be done using either `adapt` or `train`, although `train` is generally the best option, since it typically has access to more efficient training algorithms. Incremental training can only be done with `adapt`; `train` can only perform batch training.

Let's begin with the static network we used in previous examples. The learning rate will be set to 0.1.

```
net = newlin([-1 1;-1 1],1,0,0.1);
```

```
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For batch training of a static network with `adapt`, the input vectors must be placed in one matrix of concurrent vectors.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

When we call `adapt`, it will invoke `trains` (which is the default adaptation function for the linear network) and `learnwh` (which is the default learning function for the weights and biases). Therefore, Widrow-Hoff learning is used.

```
[net,a,e,pf] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all of the training set has been presented. If we display the weights we find:

```
»net.IW{1,1}
  ans = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

This is different that the result we had after one pass of `adapt` with incremental updating.

Now let's perform the same batch training using `train`. Since the Widrow-Hoff rule can be used in incremental or batch mode, it can be invoked by `adapt` or `train`. There are several algorithms that can only be used in batch mode (e.g., Levenberg-Marquardt), and so these algorithms can only be invoked by `train`.

The network will be set up in the same way.

```
net = newlin([-1 1;-1 1],1,0,0.1);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For this case, the input vectors can either be placed in a matrix of concurrent vectors or in a cell array of sequential vectors. Within `train` any cell array of sequential vectors is converted to a matrix of concurrent vectors. This is

because the network is static, and because `train` always operates in the batch mode. Concurrent mode operation is generally used whenever possible, because it has a more efficient MATLAB implementation.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

Now we are ready to train the network. We will train it for only one epoch, since we used only one pass of `adapt`. The default training function for the linear network is `trainc`, and the default learning function for the weights and biases is `learnwh`, so we should get the same results that we obtained using `adapt` in the previous example, where the default adaptation function was `trains`.

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1}.learnParam.lr = 0.1;
net.trainParam.epochs = 1;
net = train(net,P,T);
```

If we display the weights after one epoch of training we find:

```
»net.IW{1,1}
  ans = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

This is the same result we had with the batch mode training in `adapt`. With static networks, the `adapt` function can implement incremental or batch training depending on the format of the input data. If the data is presented as a matrix of concurrent vectors, batch training will occur. If the data is presented as a sequence, incremental training will occur. This is not true for `train`, which always performs batch training, regardless of the format of the input.

Batch Training With Dynamic Networks

Training static networks is relatively straightforward. If we use `train` the network is trained in the batch mode and the inputs is converted to concurrent vectors (columns of a matrix), even if they are originally passed as a sequence (elements of a cell array). If we use `adapt`, the format of the input determines the method of training. If the inputs are passed as a sequence, then the network is trained in incremental mode. If the inputs are passed as concurrent vectors, then batch mode training is used.

With dynamic networks, batch mode training is typically done with `train` only, especially if only one training sequence exists. To illustrate this, let's consider again the linear network with a delay. We use a learning rate of 0.02 for the training. (When using a gradient descent algorithm, we typically use a smaller learning rate for batch mode training than incremental training, because all of the individual gradients are summed together before determining the step change to the weights.)

```
net = newlin([-1 1],1,[0 1],0.02);
net.IW{1,1}=[0 0];
net.biasConnect=0;
net.trainParam.epochs = 1;
Pi = {1};
P = {2 3 4};
T = {3 5 6};
```

We want to train the network with the same sequence we used for the incremental training earlier, but this time we want to update the weights only after all of the inputs are applied (batch mode). The network is simulated in sequential mode because the input is a sequence, but the weights are updated in batch mode.

```
net=train(net,P,T,Pi);
```

The weights after one epoch of training are

```
»net.IW{1,1}
ans = 0.9000    0.6200
```

These are different weights than we would obtain using incremental training, where the weights would be updated three times during one pass through the training set. For batch training the weights are only updated once in each epoch.

Summary

The inputs to a neuron include its bias and the sum of its weighted inputs (using the inner product). The output of a neuron depends on the neuron's inputs and on its transfer function. There are many useful transfer functions.

A single neuron cannot do very much. However, several neurons can be combined into a layer or multiple layers that have great power. Hopefully this toolbox makes it easy to create and understand such large networks.

The architecture of a network consists of a description of how many layers a network has, the number of neurons in each layer, each layer's transfer function, and how the layers connect to each other. The best architecture to use depends on the type of problem to be represented by the network.

A network effects a computation by mapping input values to output values. The particular mapping problem to be performed fixes the number of inputs, as well as the number of outputs for the network.

Aside from the number of neurons in a network's output layer, the number of neurons in each layer is up to the designer. Except for purely linear networks, the more neurons in a hidden layer, the more powerful the network.

If a linear mapping needs to be represented linear neurons should be used. However, linear networks cannot perform any nonlinear computation. Use of a nonlinear transfer function makes a network capable of storing nonlinear relationships between input and output.

A very simple problem can be represented by a single layer of neurons. However, single-layer networks cannot solve certain problems. Multiple feed-forward layers give a network greater freedom. For example, any reasonable function can be represented with a two-layer network: a sigmoid layer feeding a linear output layer.

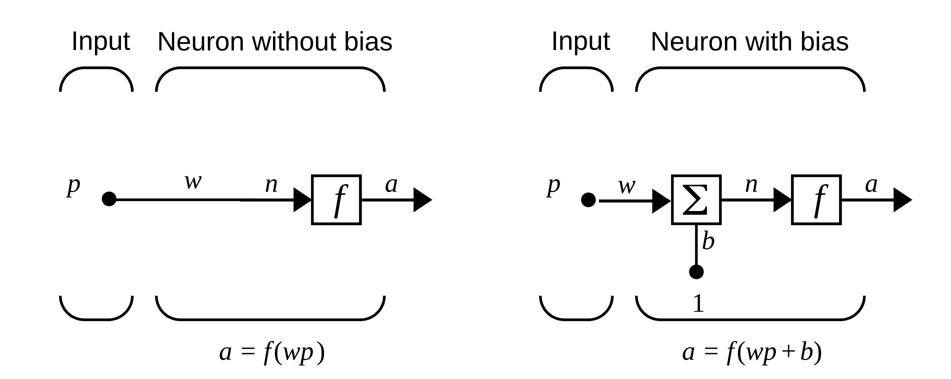
Networks with biases can represent relationships between inputs and outputs more easily than networks without biases. (For example, a neuron without a bias will always have a net input to the transfer function of zero when all of its inputs are zero. However, a neuron with a bias can learn to have any net transfer function input under the same conditions by learning an appropriate value for the bias.)

Feed-forward networks cannot perform temporal computation. More complex networks with internal feedback paths are required for temporal behavior.

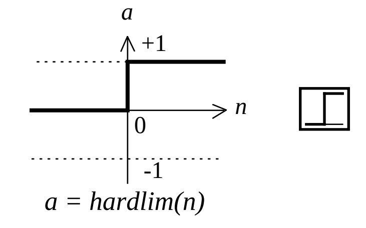
If several input vectors are to be presented to a network, they may be presented sequentially or concurrently. Batching of concurrent inputs is computationally more efficient and may be what is desired in any case. The matrix notation used in MATLAB makes batching simple.

Figures and Equations

Simple Neuron

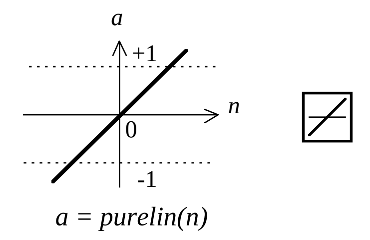


Hard Limit Transfer Function



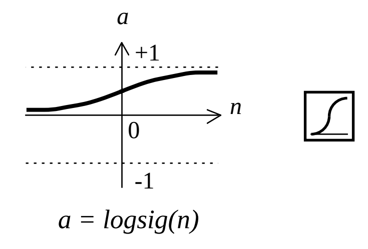
Hard-Limit Transfer Function

Purelin Transfer Function



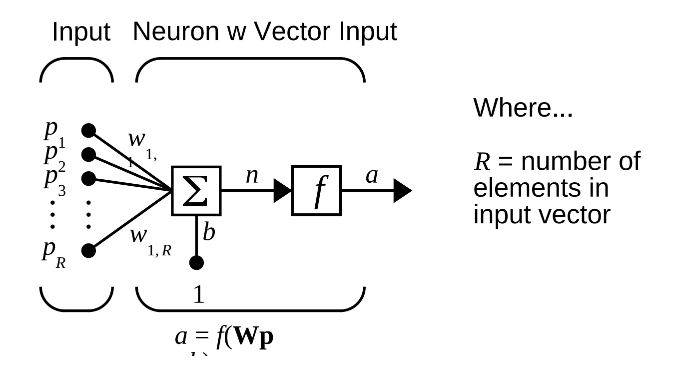
Linear Transfer Function

Log Sigmoid Transfer Function



Log-Sigmoid Transfer Function

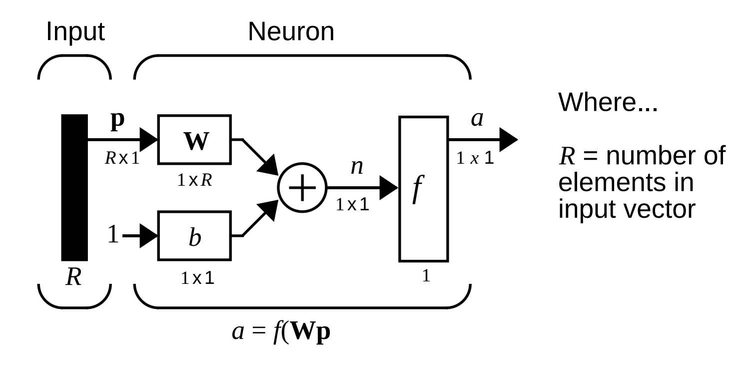
Neuron With Vector Input



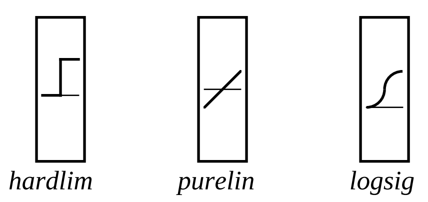
Net Input

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

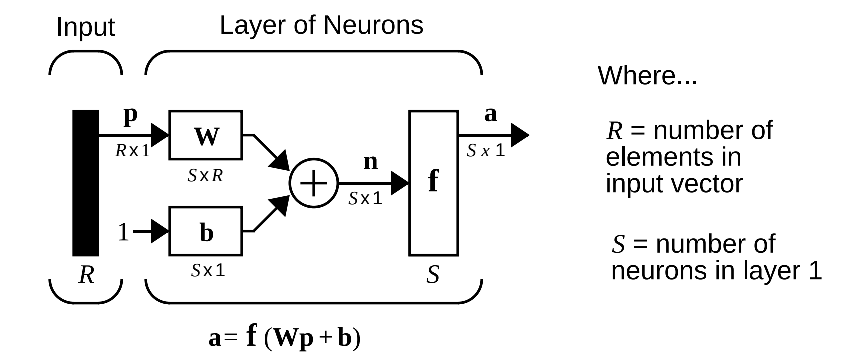
Single Neuron Using Abbreviated Notation



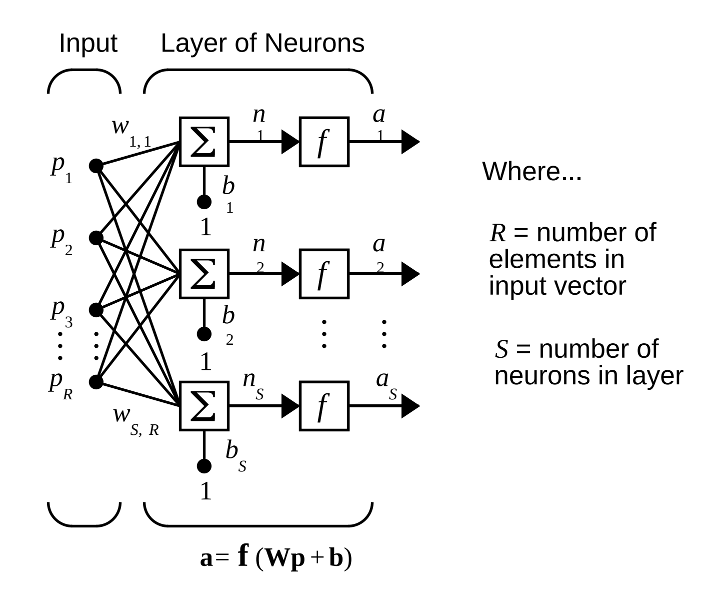
Icons for Transfer Functions



Layer of Neurons



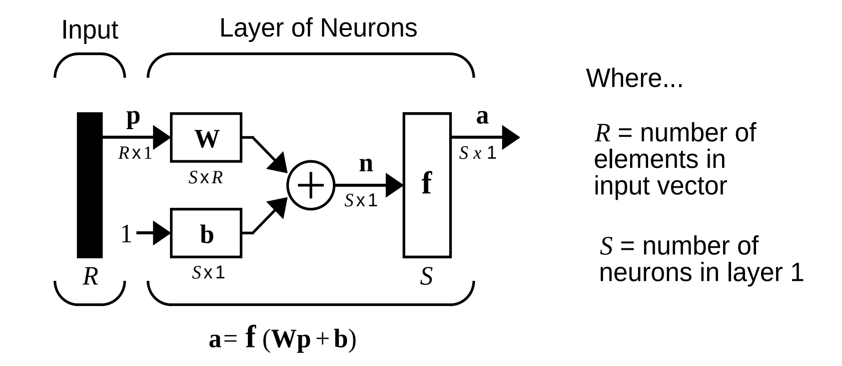
A Layer of Neurons



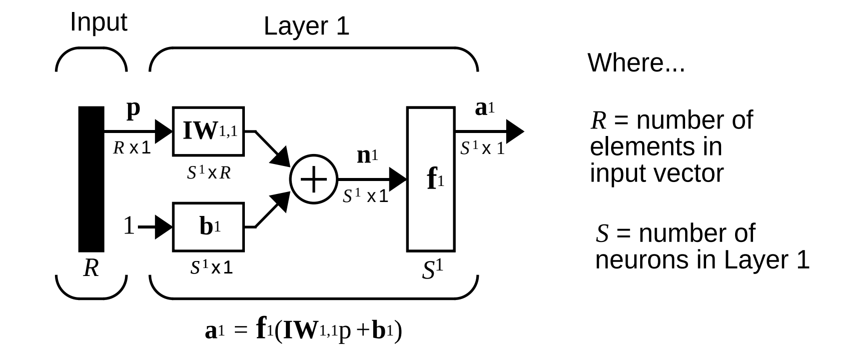
Weight Matrix

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

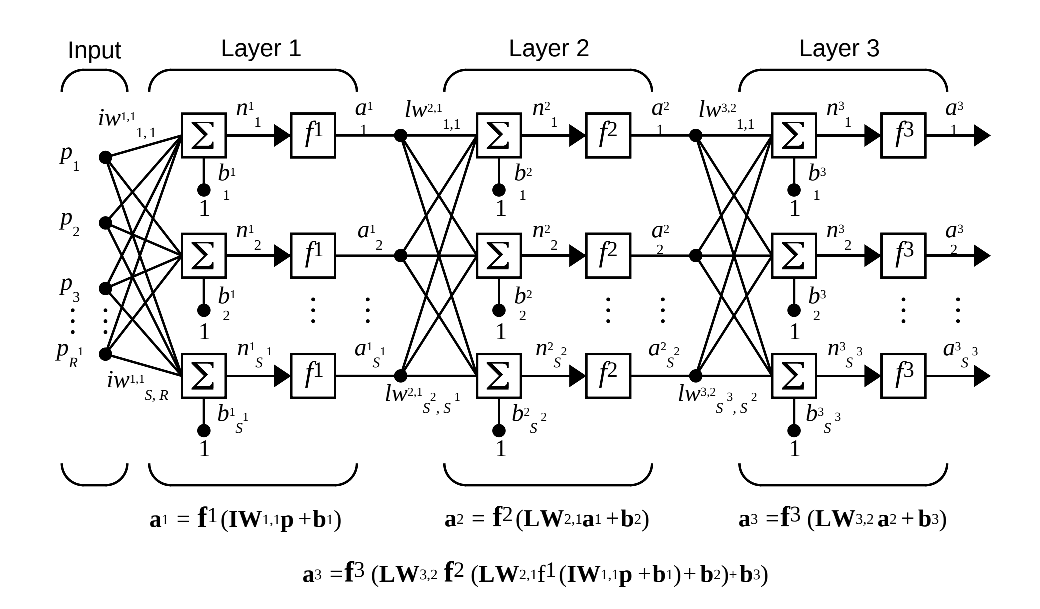
Layer of Neurons, Abbreviated Notation



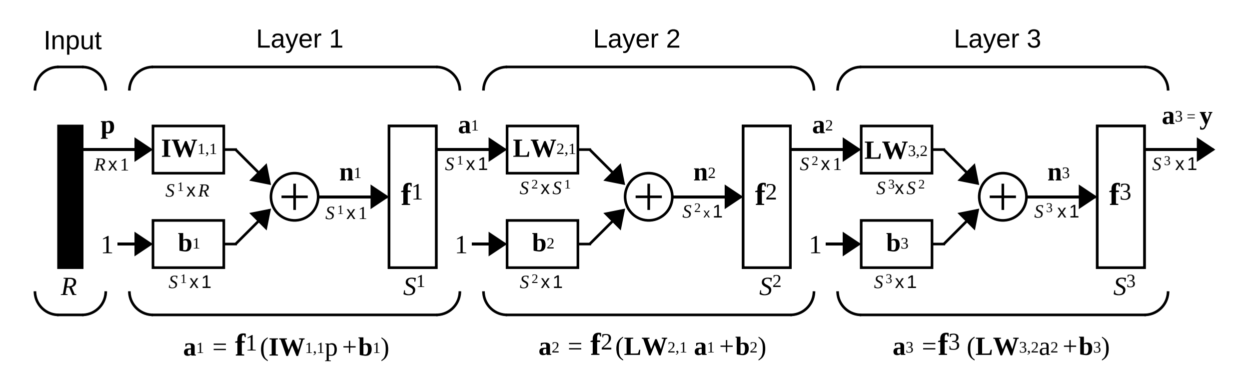
Layer of Neurons Showing Indices



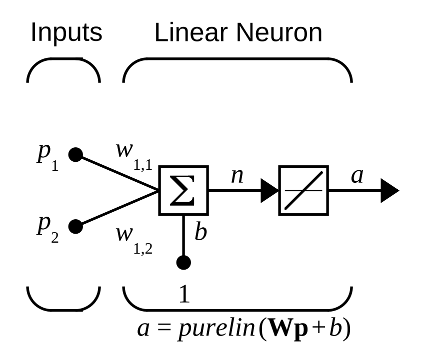
Three Layers of Neurons



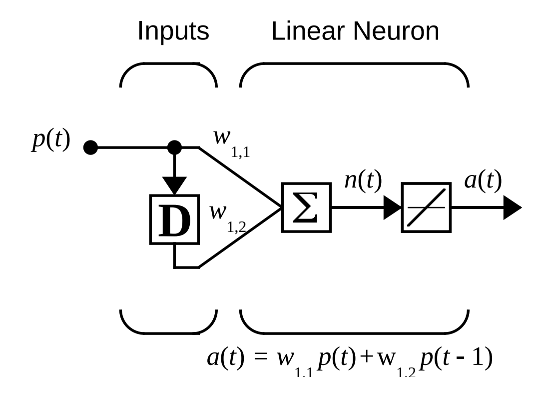
Three Layers, Abbreviated Notation



Linear Neuron With Two-Element Vector Input



Dynamic Network With One Delay



Perceptrons

Introduction 3-2
Important Perceptron Functions 3-3

Neuron Model 3-4

Perceptron Architecture 3-6

Creating a Perceptron (newp) 3-7
Simulation (sim) 3-8
Initialization (init) 3-9

Learning Rules 3-12

Perceptron Learning Rule (learnp) 3-13

Training (train) 3-16

Limitations and Cautions 3-21
Outliers and the Normalized Perceptron Rule 3-21

Graphical User Interface 3-23
Introduction to the GUI 3-23
Create a Perceptron Network (nntool) 3-23
Train the Perceptron 3-27
Export Perceptron Results to Workspace 3-29
Clear Network/Data Window 3-30
Importing from the Command Line 3-30
Save a Variable to a File and Load It Later 3-31

Summary 3-33
Figures and Equations 3-33
New Functions 3-36

Introduction

This chapter has a number of objectives. First we want to introduce you to learning rules, methods of deriving the next changes that might be made in a network, and training, a procedure whereby a network is actually adjusted to do a particular job. Along the way we discuss a toolbox function to create a simple perceptron network, and we also cover functions to initialize and simulate such networks. We use the perceptron as a vehicle for tying these concepts together.

Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

In this chapter we define what we mean by a learning rule, explain the perceptron network and its learning rule, and tell you how to initialize and simulate perceptron networks.

The discussion of perceptron in this chapter is necessarily brief. For a more thorough discussion, see Chapter 4 “Perceptron Learning Rule” of [HDB1996], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

You also may want to refer to the original book on the perceptron, Rosenblatt, F., *Principles of Neurodynamics,* Washington D.C.: Spartan Press, 1961. [Rose61].

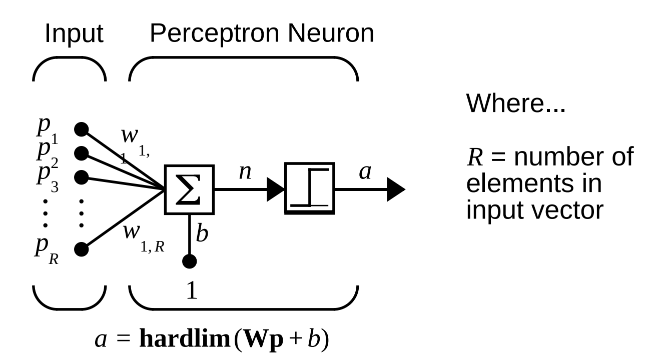
Important Perceptron Functions

Entering `help percept` at the MATLAB command line displays all the functions that are related to perceptrons.

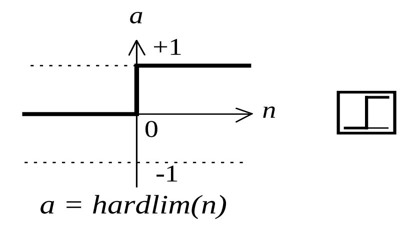
Perceptron networks can be created with the function `newp`. These networks can be initialized, simulated and trained with the `init`, `sim` and `train`. The following material describes how perceptrons work and introduces these functions.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



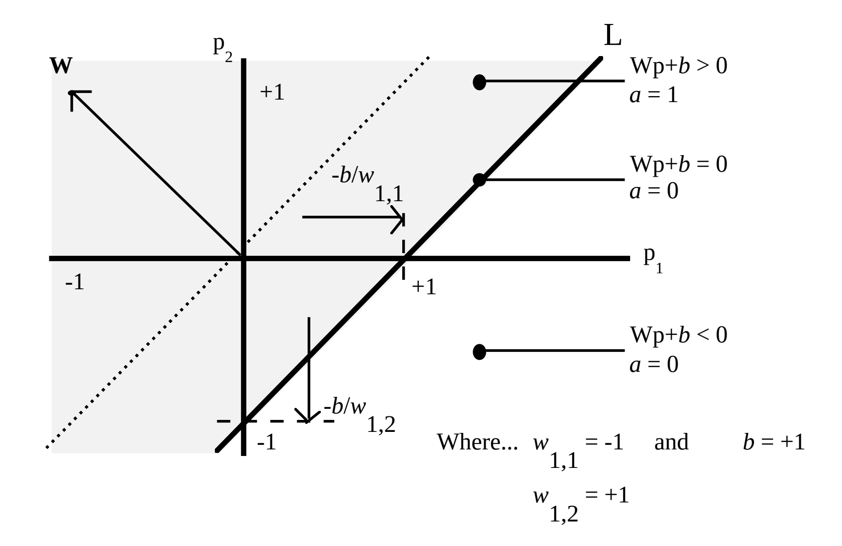
Each external input is weighted with an appropriate weight w_{1j}, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$, is shown below.



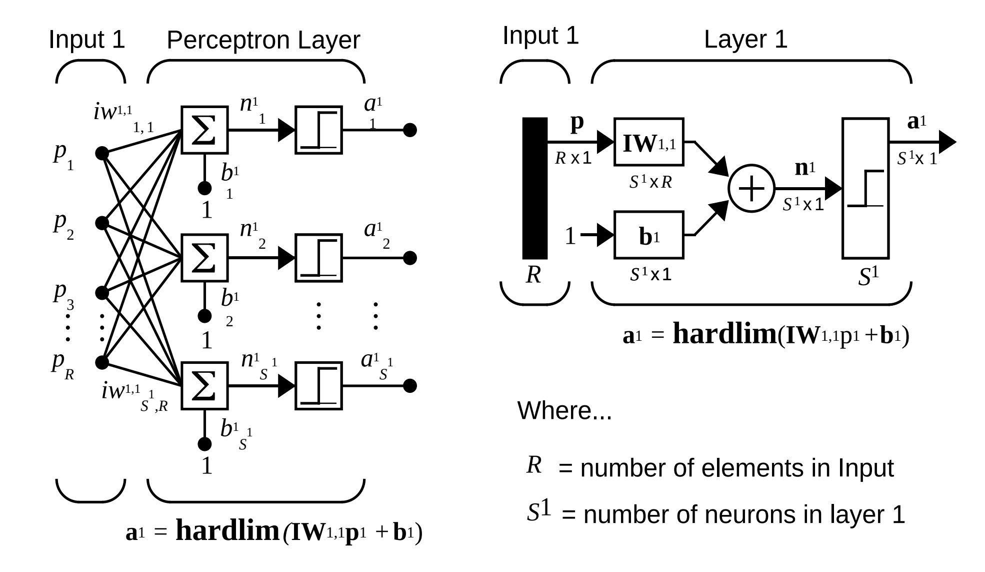
Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$. This line is perpendicular to the weight matrix **W** and shifted according to the bias b. Input vectors above and to the left of the line L will result in a net input greater than 0; and therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. The dividing line can be oriented and moved anywhere to classify the input space as desired by picking the weight and bias values.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin as shown in the plot above.

You may want to run the demonstration program `nnd4db`. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights $w_{i,j}$ as shown below in two forms. As before, the network indices *i* and *j* indicate that $w_{i,j}$ is the strength of the connection from the *j*th input to the *i*th neuron.



The perceptron learning rule that we will describe shortly is capable of training only a single layer. Thus, here we will consider only one-layer networks. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed later in this chapter in the "Limitations and Cautions" section.

Creating a Perceptron (newp)

A perceptron can be created with the function `newp`

```
net = newp(PR, S)
```

where input arguments:

`PR` is an R-by-2 matrix of minimum and maximum values for R input elements.

`S` is the number of neurons.

Commonly the `hardlim` function is used in perceptrons, so it is the default.

The code below creates a peceptron network with a single one-element input vector and one neuron. The range for the single element of the single input vector is [0 2].

```
net = newp([0 2],1);
```

We can see what network has been created by executing the following code

```
inputweights = net.inputweights{1,1}
```

which yields:

```
inputweights =
        delays: 0
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: [1 1]
      userdata: [1x1 struct]
     weightFcn: 'dotprod'
```

Note that the default learning function is `learnp`, which is discussed later in this chapter. The net input to the `hardlim` transfer function is `dotprod`, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

Also note that the default initialization function, `initzero`, is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: 1
      userdata: [1x1 struct]
```

We can see that the default initialization for the bias is also 0.

Simulation (sim)

To show how `sim` works we examine a simple problem.

Suppose we take a perceptron with a single two-element input vector, like that discussed in the *decision boundary* figure. We define the network with

```
net = newp([-2 2;-2 +2],1);
```

As noted above, this gives us zero weights and biases, so if we want a particular set other than zeros, we have to create them. We can set the two weights and the one bias to -1, 1 and 1 as they were in the decision boundary figure with the following two lines of code.

```
net.IW{1,1}= [-1 1];
net.b{1} = [1];
```

To make sure that these parameters were set correctly, we check them with

```
net.IW{1,1}
ans =
    -1     1
net.b{1}
ans =

     1
```

Now let us see if the network responds to two signals, one on each side of the perceptron boundary.

```
p1 = [1;1];
```

```
a1 = sim(net,p1)
a1 =

     1
```

and for

```
p2 = [1;-1]
a2 = sim(net,p2)
a2 =

     0
```

Sure enough, the perceptron classified the two inputs correctly.

Note that we could present the two inputs in a sequence and get the outputs in a sequence as well.

```
p3 = {[1;1] [1;-1]};
a3 = sim(net,p3)
a3 =

    [1]    [0]
```

You may want to read more about `sim` in “Advanced Topics” in Chapter 12.

Initialization (init)

You can use the function `init` to reset the network weights and biases to their original values. Suppose, for instance that you start with the network

```
net = newp([-2 2;-2 +2],1);
```

Now check its weights with

```
wts = net.IW{1,1}
```

which gives, as expected,

```
wts =

     0     0
```

In the same way, you can verify that the bias is 0 with

```
bias = net.b{1}
```

which gives

```
bias =

     0
```

Now set the weights to the values 3 and 4 and the bias to the value 5 with

```
net.IW{1,1} = [3,4];
net.b{1} = 5;
```

Recheck the weights and bias as shown above to verify that the change has been made. Sure enough,

```
wts =
     3     4
bias =

     5
```

Now use `init` to reset the weights and bias to their original values.

```
net = init(net);
```

We can check as shown above to verify that.

```
wts =
     0     0
bias =

     0
```

We can change the way that a perceptron is initialized with `init`. For instance, we can redefine the network input weights and bias `initFcns` as `rands`, and then apply `init` as shown below.

```
net.inputweights{1,1}.initFcn = 'rands';
net.biases{1}.initFcn = 'rands';
net = init(net);
```

Now check on the weights and bias.

```
wts =
    0.2309    0.5839
biases =
```

```
-0.1106
```

We can see that the weights and bias have been given random numbers.

You may want to read more about `init` in “Advanced Topics” in Chapter 12.

Learning Rules

We define a *learning rule* as a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules in this toolbox fall into two broad categories: supervised learning, and unsupervised learning.

In *supervised learning*, the learning rule is provided with a set of examples (the *training set*) of proper network behavior

$$\{\mathbf{p}_1,\mathbf{t}_1\}, \{\mathbf{p}_2,\mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q,\mathbf{t}_Q\}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category.

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization.

As noted, the perceptron discussed in this chapter is trained with supervised learning. Hopefully, a network that produces the right output for a particular input will be obtained.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1\mathbf{t}_1,\mathbf{p}_2\mathbf{t}_1,\ldots, \mathrm{p}_Q\mathbf{t}_Q$$

where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response **a,** and the target vector **t**. The *perceptron learning rule* `learnp` calculates desired changes to the perceptron's weights and biases given an input vector **p**, and the associated error **e**. The target vector **t** must contain values of either 0 or 1, as perceptrons (with `hardlim` transfer functions) can only output such values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector **w** to point toward input vectors to be classified as 1, and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to **w**, and which properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct ($\mathbf{a} = \mathbf{t}$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$), then the weight vector **w** is not altered.

CASE 2. If the neuron output is 0 and should have been 1 ($\mathbf{a} = 0$ and $\mathbf{t} = 1$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 ($\mathbf{a} = 1$and $\mathbf{t} = 0$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = -1$), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector is classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$, and the change to be made to the weight vector $\Delta\mathbf{w}$:

CASE 1. If $\mathbf{e} = 0$, then make a change $\Delta\mathbf{w}$ equal to 0.

CASE 2. If $\mathbf{e} = 1$, then make a change $\Delta\mathbf{w}$ equal to $\mathbf{p}^{T}$.

CASE 3. If $\mathbf{e} = -1$, then make a change $\Delta\mathbf{w}$ equal to $-\mathbf{p}^{T}$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t - a)\mathbf{p}^{T} = e\mathbf{p}^{T}$$

We can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - a)(1) = e$$

For the case of a layer of neurons we have:

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^{T} = \mathbf{e}(\mathbf{p})^{T} \text{ and}$$

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = \mathbf{E}$$

The Perceptron Learning Rule can be summarized as follows

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^{T} \text{ and}$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$.

Now let us try a simple example. We start with a single neuron having a input vector with just two elements.

```
net = newp([-2 2;-2 +2],1);
```

To simplify matters we set the bias equal to 0 and the weights to 1 and -0.8.

```
net.b{1} =  [0];
w = [1 -0.8];
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];
t = [1];
```

We can compute the output and error with

```
a = sim(net,p)
a =
     0
e = t-a
e =
     1
```

and finally use the function learnp to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[])
dw =
     1     2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Note that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are “linearly separable.” The objects to be classified in such cases can be separated by a single line.

You might want to try demo `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

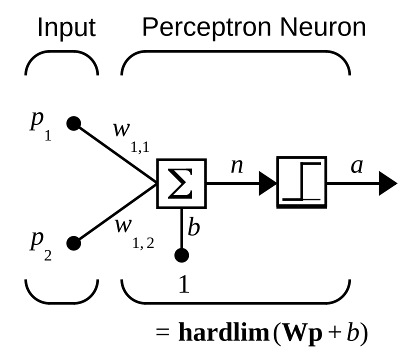
Training (train)

If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traverse through all of the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function train proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. The new values of **W** and **b** must be checked by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully it can be trained further by again calling `train` with the new weights and biases for more training passes, or the problem can be analyzed to see if it is a suitable problem for the perceptron. Problems which are not solvable by the perceptron network are discussed in the "Limitations and Cautions" section.

To illustrate the training procedure, we will work through a simple problem. Consider a one neuron perceptron with a single vector input having two elements.



This network, and the problem we are about to consider are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996].

Let us suppose we have the following classification problem and would like to solve it with our single vector input, two-element perceptron network.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias. We denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, we have the initial values, $\mathbf{W}(0)$ and $b(0)$.

$$\mathbf{W}(0) = \begin{bmatrix}0 & 0\end{bmatrix} \qquad b(0) = 0$$

We start by calculating the perceptron's output a for the first input vector $\mathbf{p_1}$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix}0 & 0\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output a does not equal the target value t_1, so we use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)\begin{bmatrix}2 & 2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix}$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the Perceptron update rules shown previously.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = \begin{bmatrix}0 & 0\end{bmatrix} + \begin{bmatrix}-2 & -2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix} = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, $\mathbf{p}_2$. The output is calculated below.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2 \\ -2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix}-2 & -2\end{bmatrix}$ and $b(2) = b(1) = -1$

We can continue in this fashion, presenting $\mathbf{p}_3$ next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values: $\mathbf{W}(4) = \begin{bmatrix}-3 & -1\end{bmatrix}$ and $b(4) = 0$. To determine if we obtained a satisfactory solution, we must make one pass through all input vectors to see if they all produce the desired target values. This is not true for the 4th input, but the algorithm does converge on the 6th presentation of an input. The final values are:

$\mathbf{W}(6) = \begin{bmatrix}-2 & -3\end{bmatrix}$ and $b(6) = 1$

This concludes our hand calculation. Now, how can we do this using the `train` function?

The following code defines a perceptron like that shown in the previous figure, with initial weights and bias values of 0.

```
net = newp([-2 2;-2 +2],1);
```

Now consider the application of a single input.

```
p =[2; 2];
```

```
having the target
```

```
t =[0];
```

Now set `epochs` to 1, so that `train` will go through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w =
    -2    -2
```

```
b =
     -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values [-2 -2] and -1, just as we hand calculated.

We now apply the second input vector $\mathbf{p}_2$. The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0 and the change will be zero. We could proceed in this way, starting from the previous result and applying a new input vector time after time. But we can do this job automatically with `train`.

Now let's apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = newp([-2 2;-2 +2],1);
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]
t =[0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w =
     -3     -1
b =
      0
```

Note that this is the same result as we got previously by hand. Finally simulate the trained network for each of the inputs.

```
a = sim(net,p)
a =
    [0]    [0]    [1]    [1]
```

The outputs do not yet equal the targets, so we need to train the network for more than one pass. We will try four epochs. This run gives the following results.

```
TRAINC, Epoch 0/20
TRAINC, Epoch 3/20
TRAINC, Performance goal met.
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As we know from our hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w =
    -2    -3
b =
     1
```

The simulated output and errors for the various inputs are

```
a =
             0          1.00              0          1.00
error = [a(1)-t(1) a(2)-t(2) a(3)-t(3) a(4)-t(4)]
error =
             0             0              0             0
```

Thus, we have checked that the training procedure was successful. The network converged and produces the correct target outputs for the four input vectors.

Note that the default training function for networks created with `newp` is `trains`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented can not be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You may want to try various demonstration programs. For instance, `demop1` illustrates classification and training of a simple perceptron.

Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of adapt in this way guarantees that any linearly separable problem is solved in a finite number of training presentations. Perceptrons can also be trained with the function `train`, which is presented in the next chapter. When `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) due to the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. Note, however, that it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try `demop6`. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996].

Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try `demop4` to see how an outlier affects the training.

By changing the perceptron learning rule slightly, training times can be made insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

As shown above, the larger an input vector **p**, the larger its effect on the weight vector **w**. Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t-a)\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|} = e\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnpn`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces number of epochs considerably if there are outlier input vectors. You might try `demop5` to see how this normalized training rule works.

Graphical User Interface

Introduction to the GUI

The graphical user interface (GUI) is designed to be simple and user friendly, but we will go through a simple example to get you started.

In what follows you bring up a GUI **Network/Data Manager** window. This window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, you might "export" the GUI results to the (command line) workspace. Similarly you may want to "import" results from the command line workspace to the GUI.

Once the **Network/Data Manager** is up and running, you can create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI.

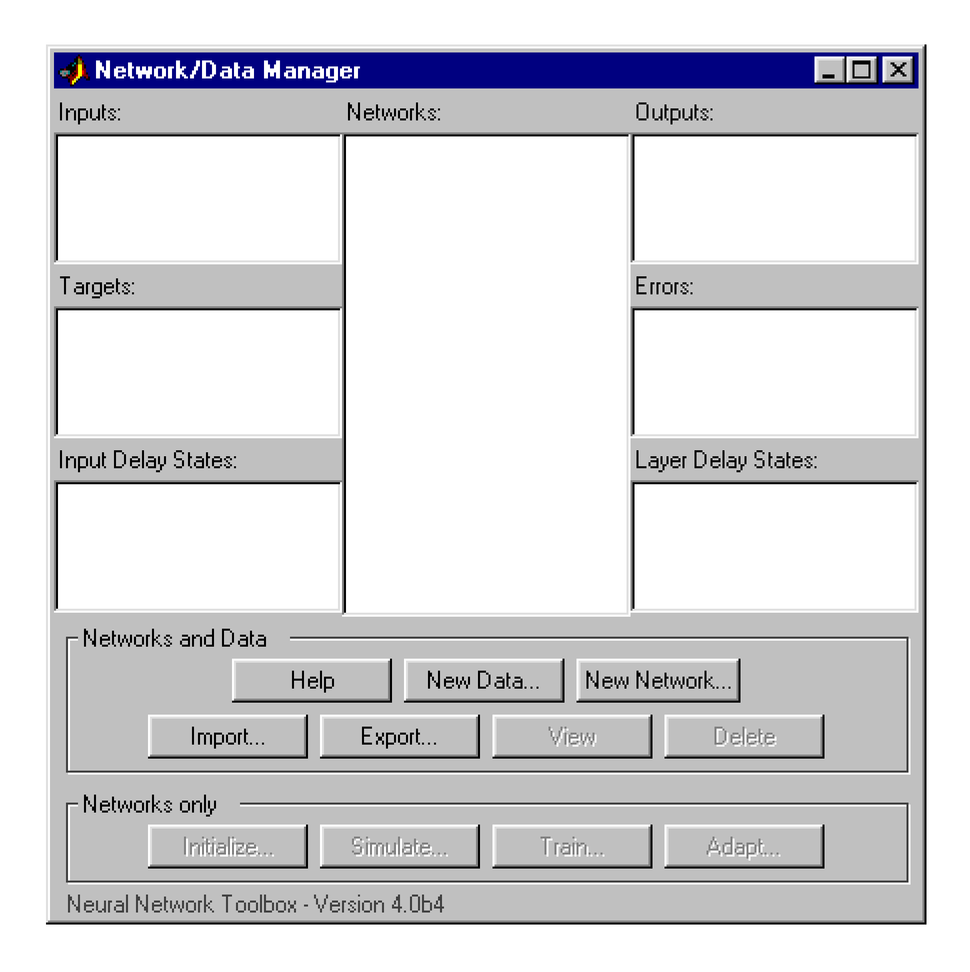
The following example deals with a perceptron network. We go through all the steps of creating a network and show you what you might expect to see as you go along.

Create a Perceptron Network (nntool)

We create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. We call the network `ANDNet`. Once created, the network will be trained. We can then save the network, its output, etc., by "exporting" it to the command line.

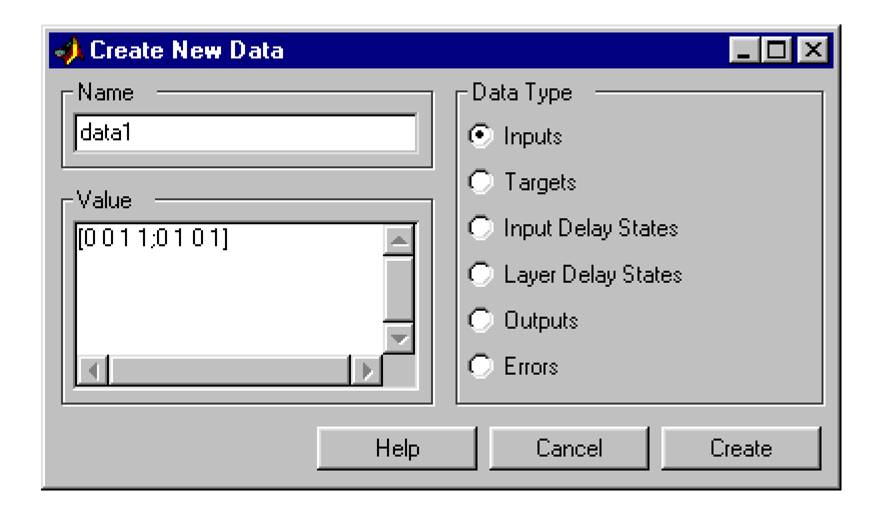
Input and target

To start, type `nntool`. The following window appears.



Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists.

First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this:

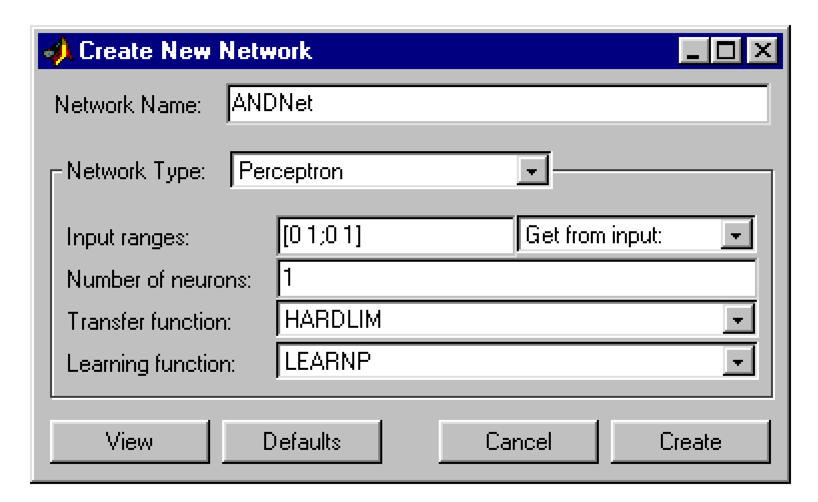


Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input.

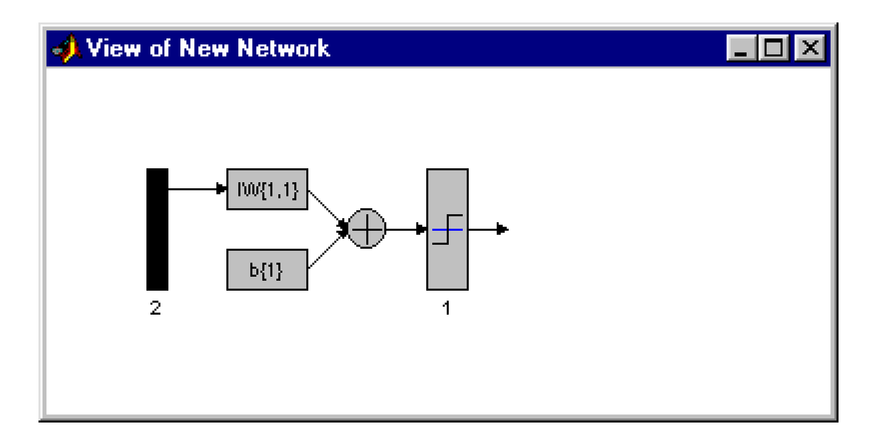
Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/ Data Manager** window that you now have t as a target as well as the previous p as an input.

Create Network

Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like:



Next you might look at the network by clicking on **View**. For example:

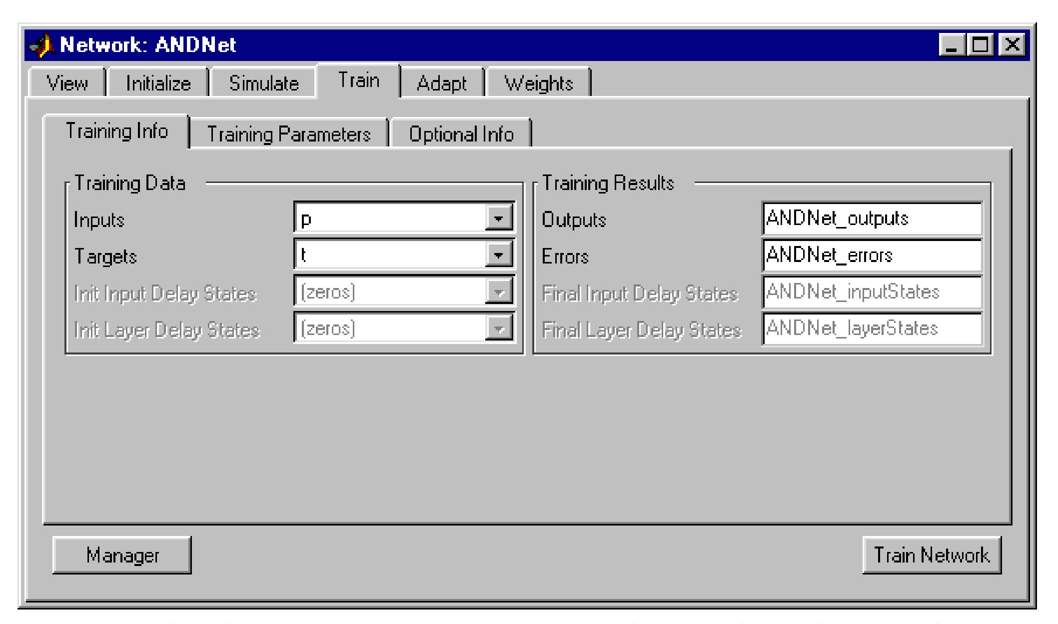


This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted.

Now click **Create** to generate the network. You will get back the **Network/ Data Manager** window. Note that `ANDNet` is now listed as a network.

Train the Perceptron

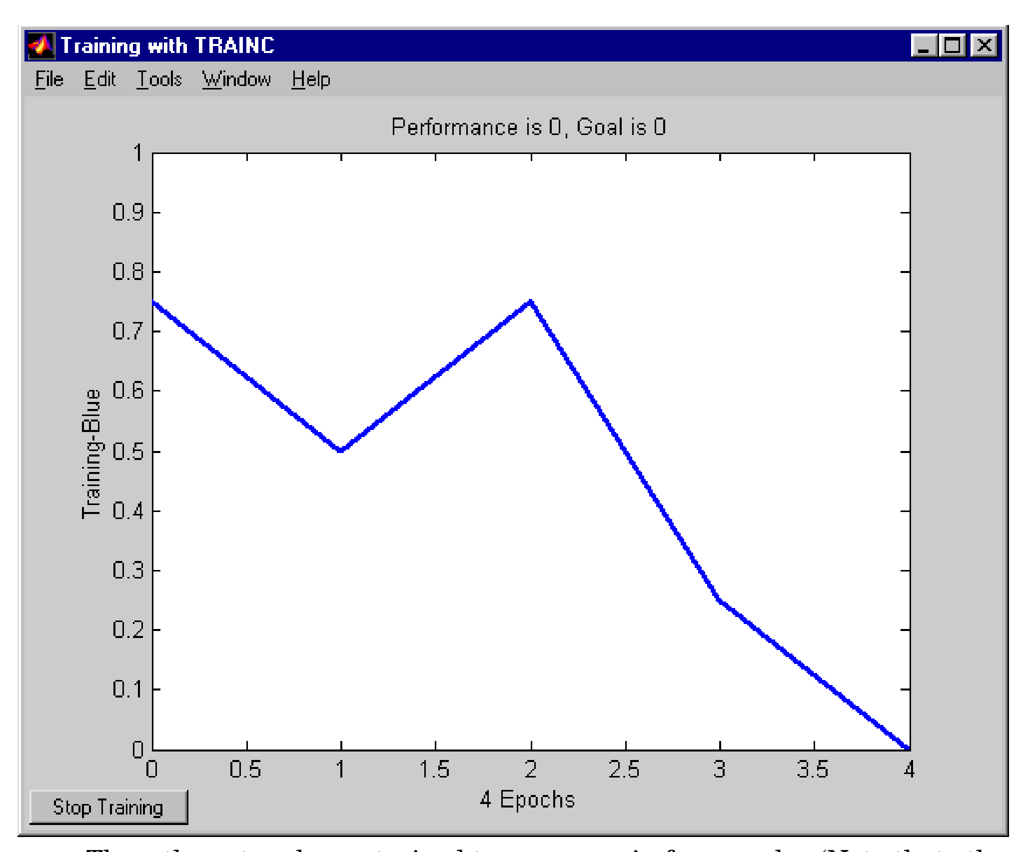
To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:



Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line.

While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want.

Now click **Train Network** to train the perceptron network. You will see the following training results.



Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.)

You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate)**. This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`.

Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value

```
[0 0 0 1]
```

Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.

Export Perceptron Results to Workspace

To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be

```
who
Your variables are:
ANDNet_errors       ANDNet_outputs
```

You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following

```
ANDNet_outputs =
0     0     0     1
```

and

```
ANDNet_errors =
0     0     0     0.
```

You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.

Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command

```
ANDNet.iw{1,1}
```

gives

```
ans =
```

```
2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
ans =
-3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window.

Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

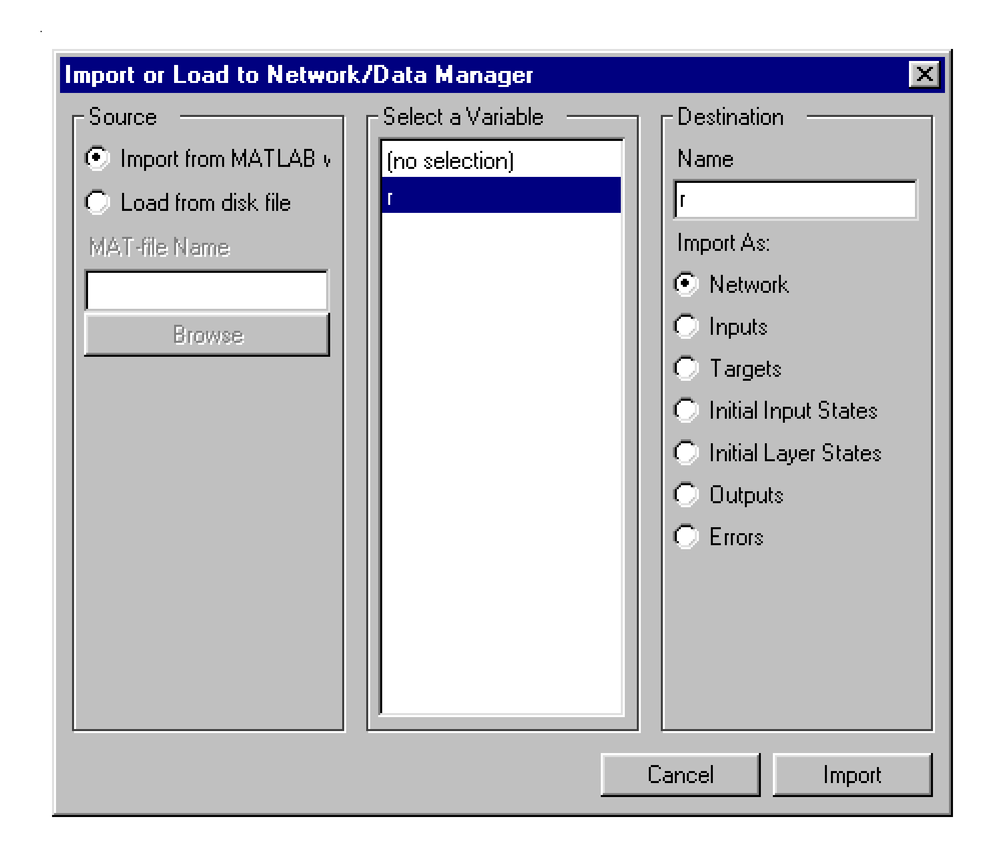
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session.

Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this



Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input.

Save a Variable to a File and Load It Later

Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile.`

Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager**

window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window.

Summary

Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem.

The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem.

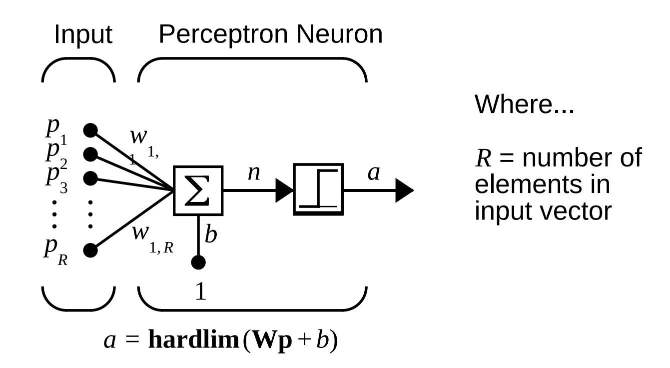
Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution.

Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors.

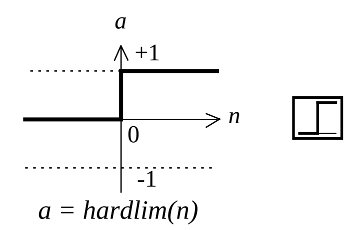
A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace.

Figures and Equations

Perceptron Neuron

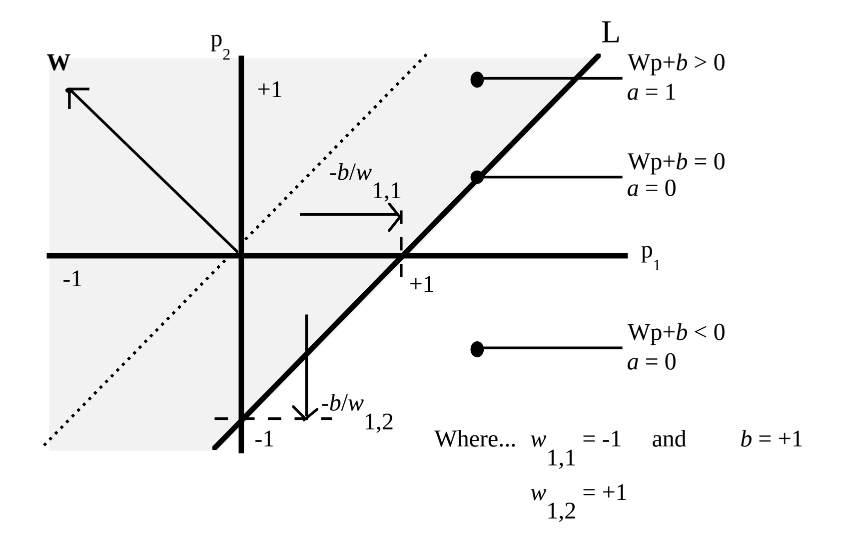


Perceptron Transfer Function, `hardlim`

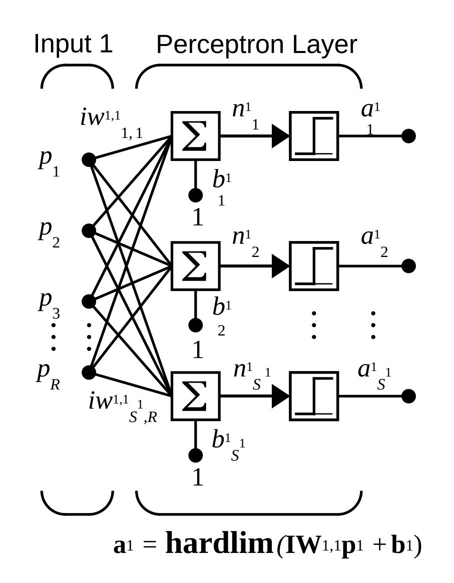


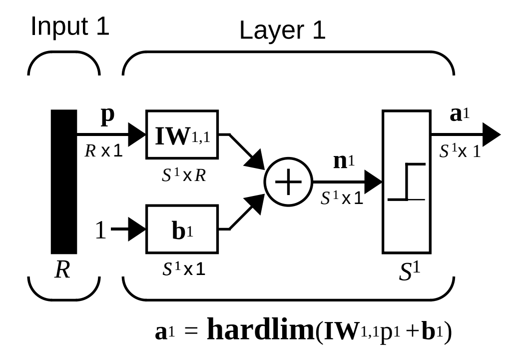
Hard-Limit Transfer Function

Decision Boundary



Perceptron Architecture





Where...

R = number of elements in Input

S^1 = number of neurons in layer 1

The Perceptron Learning Rule

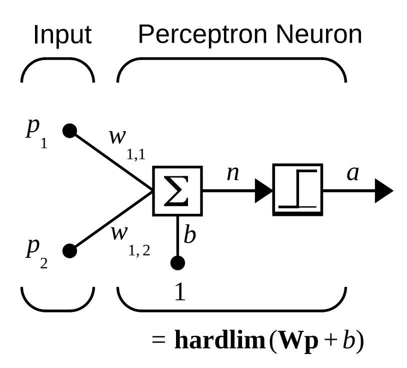
$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where

$$\mathbf{e} = \mathbf{t} - \mathbf{a}$$

One Perceptron Neuron



New Functions

This chapter introduces the following new functions.

Function	Description
`hardlim`	A hard limit transfer function
`initzero`	Zero weight/bias initialization function
`dotprod`	Dot product weight function
`newp`	Creates a new perceptron network.
`sim`	Simulates a neural network.
`init`	Initializes a neural network
`learnp`	Perceptron learning function
`learnpn`	Normalized perceptron learning function
`nntool`	Starts the Graphical User Interface (GUI)

4

Linear Filters

Introduction 4-2

Neuron Model 4-3

Network Architecture 4-4
Creating a Linear Neuron (newlin) 4-4

Mean Square Error 4-8

Linear System Design (newlind) 4-9

Linear Networks with Delays 4-10
Tapped Delay Line 4-10
Linear Filter 4-10

LMS Algorithm (learnwh) 4-13

Linear Classification (train) 4-15

Limitations and Cautions 4-18
Overdetermined Systems 4-18
Underdetermined Systems 4-18
Linearly Dependent Vectors 4-18
Too Large a Learning Rate 4-19

Summary 4-20
Figures and Equations 4-21
New Functions 4-25

Introduction

The linear networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here we will design a linear network that, when presented with a set of given input vectors, produces outputs of corresponding target vectors. For each input vector we can calculate the network's output vector. The difference between an output vector and its target vector is the error. We would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, we can calculate a linear network directly, such that its error is a minimum for the given input vectors and targets vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, we can always train the network to have a minimum error by using the Least Mean Squares (Widrow-Hoff) algorithm.

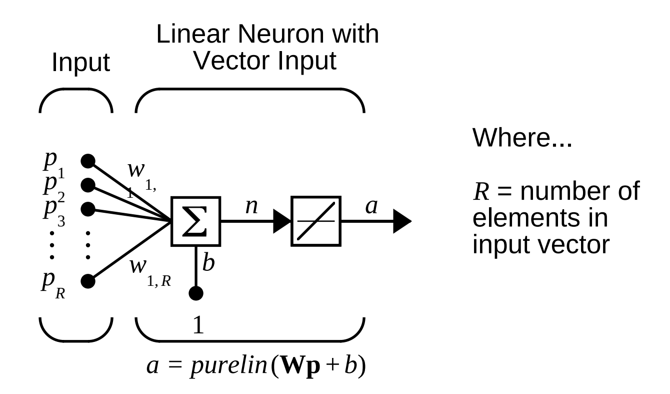
Note that the use of linear filters in adaptive systems is discussed in Chapter 10.

This chapter introduces `newlin`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

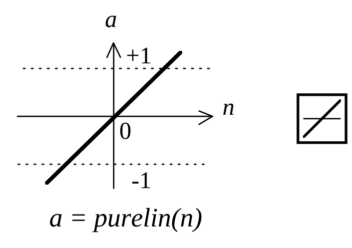
You can type `help linnet` to see a list of linear network functions, demonstrations, and applications.

Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, which we will give the name `purelin`.



Linear Transfer Function

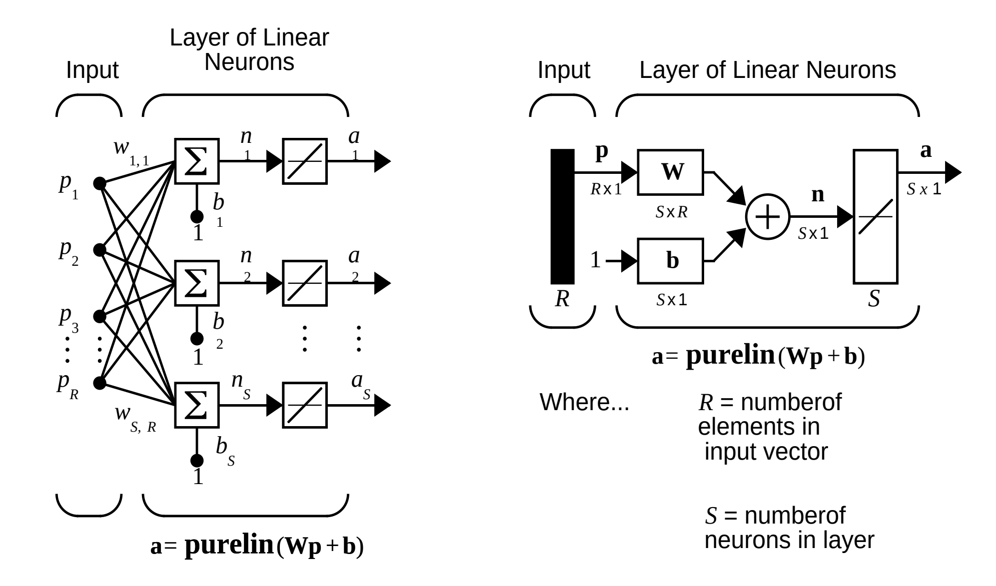
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Network Architecture

The linear network shown below has one layer of S neurons connected to R inputs through a matrix of weights **W**.

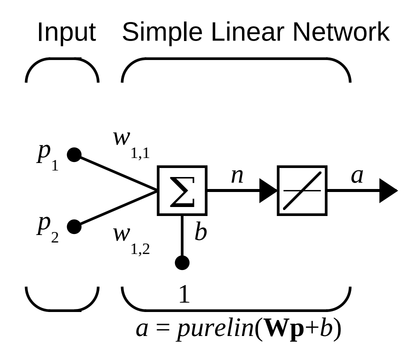


Note that the figure on the right defines an S-length output vector **a**.

We have shown a single-layer linear network. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Creating a Linear Neuron (newlin)

Consider a single linear neuron with two inputs. The diagram for this network is shown below.

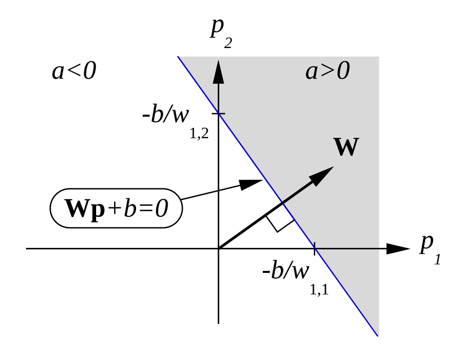


The weight matrix **W** in this case has only one row. The network output is:

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b \qquad \text{or}$$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary as shown below (adapted with thanks from [HDB96]).



Input vectors in the upper right gray area will lead to an output greater than 0. Input vectors in the lower left white area will lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear network has the same limitation as the perceptron.

We can create a network like that shown above with the command

```
net = newlin( [-1 1; -1 1],1);
```

The first matrix of arguments specify the range of the two scalar inputs. The last argument, 1, says that the network has a single output.

The network weights and biases are set to zero by default. You can see the current values with the commands

```
W = net.IW{1,1}
W =
     0     0
```

and

```
b= net.b{1}
b =
     0
```

However, you can give the weights any value that you want, such as 2 and 3 respectively, with

```
net.IW{1,1} = [2 3];
W = net.IW{1,1}
W =
     2     3
```

The bias can be set and checked in the same way.

```
net.b{1} =[-4];
b = net.b{1}
b =
     -4
```

You can simulate the linear network for a particular input vector. Try

```
p = [5;6];
```

Now you can find the network output with the function `sim`.

```
a = sim(net,p)
a =
    24
```

To summarize, you can create a linear network with `newlin`, adjust its elements as you want, and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. We want to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Ch. 10 of [HDB96].

Linear System Design (newlind)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. Specific network values for weights and biases can be obtained to minimize the mean square error by using the function `newlind`.

Suppose that the inputs and targets are

```
P = [1 2 3];
T= [2.0 4.1 5.9];
```

Now you can design a network.

```
net = newlind(P,T);
```

You can simulate the network behavior to check that the design was done properly.

```
Y = sim(net,P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

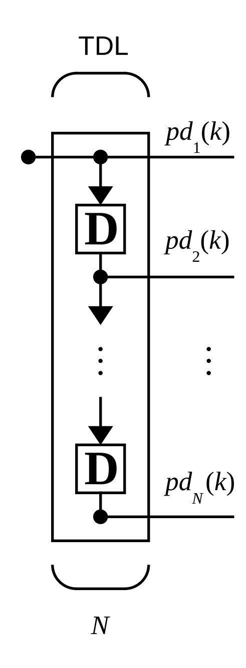
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design and plots the designed solution.

The function `newlind` can also be used to design linear networks having delays in the input. Such networks are discussed later in this chapter. First, however, we need to discuss delays.

Linear Networks with Delays

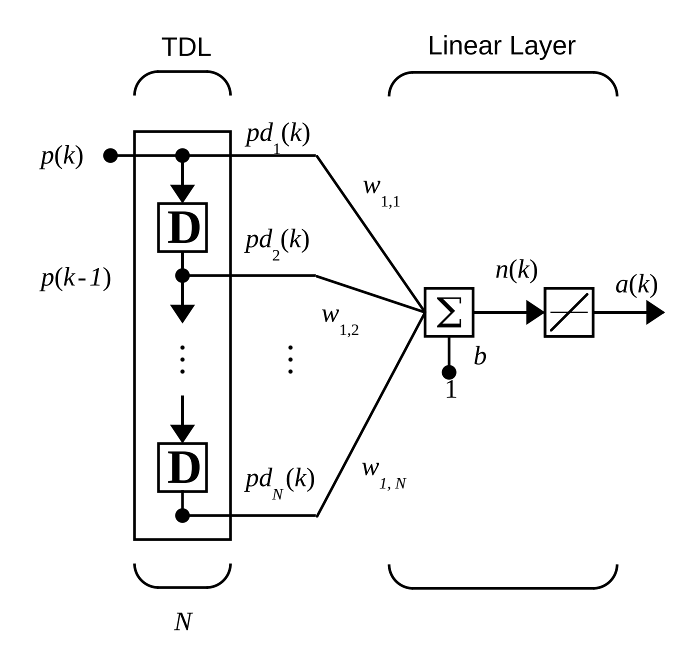
Tapped Delay Line

We need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left, and passes through N-1 delays. The output of the tapped delay line (TDL) is an N-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Linear Filter

We can combine a tapped delay line with an linear network to create the *linear* filter shown below.



The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i}a(k-i+1) + b$$

The network shown above is referred to in the digital signal-processing field as a finite impulse response (FIR) filter [WiSt85]. Let us take a look at the code that we use to generate and simulate such a specific network.

Suppose that we want a linear layer that outputs the sequence `T` given the sequence P and two initial input delay states `Pi`.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5 6 4 20 7 8};
```

You can use `newlind` to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument as shown below.

```
net = newlind(P,T,Pi);
```

Now we obtain the output of the designed network with

```
Y = sim(net,P,Pi)
```

to give

```
Y = [2.73]    [10.54]    [5.01]    [14.95]    [10.78]    [5.98]
```

As you can see, the network outputs are not exactly equal to the targets, but they are reasonably close, and in any case, the mean square error is minimized.

LMS Algorithm (learnwh)

The LMS algorithm or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If we take the partial derivative of the squared error with respect to the weights and biases at the *k*th iteration we have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1, 2, \ldots, R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial[t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - (\mathbf{Wp}(k) + b)] \text{ or}$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right]$$

Here p_i(k) is the *i*th element of the input vector at the *k*th iteration.

Similarly,

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

This can be simplified to:

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \text{ and}$$

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, the change to the weight matrix and the bias will be

$2\alpha e(k)\mathbf{p}(k)$ and $2\alpha e(k)$.

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors and α is a *learning rate*. If α is large, learning occurs quickly, but if it is too large it may lead to instability and errors may even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix $\mathbf{p}^T\mathbf{p}$ of the input vectors.

You might want to read some of Chapter 10 of [HDB96] for more information about the LMS algorithm and its convergence.

Fortunately we have a toolbox function `learnwh` that does all of the calculation for us. It calculates the change in weights as

```
dw = lr*e*p'
```

and the bias change as

```
db = lr*e
```

The constant 2, shown a few lines above, has been absorbed into the code learning rate `lr`. The function `maxlinlr` calculates this maximum stable learning rate `lr` as 0.999 * `P'*P`.

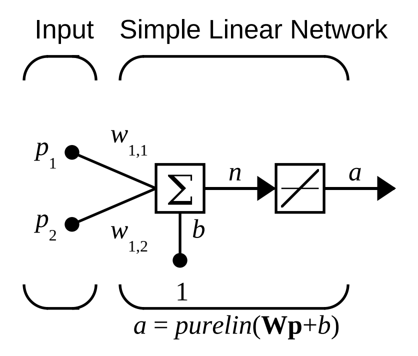
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

Linear Classification (train)

Linear networks can be trained to perform linear classification with the function `train`. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to `learnp`. Then the network is adjusted with the sum of all these corrections. We will call each pass through the input vectors an epoch. This contrasts with `adapt`, discussed in "Adaptive Filters and Adaptive Training" in Chapter 10, which adjusts weights for each input vector as it is presented.

Finally, `train` applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

To illustrate this procedure, we will work through a simple problem. Consider the linear network introduced earlier in this chapter.



Next suppose we have the classification problem presented in "Linear Filters" in Chapter 4.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Here we have four input vectors, and we would like a network that produces the output corresponding to each input vector when that vector is presented.

We will use `train` to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network will be 0 by default. We will set the error goal to 0.1 rather than accept its default of 0.

```
P = [2 1 -2 -1;2 -2 2 1];
t = [0 1 0 1];
net = newlin( [-2 2; -2 2],1);
net.trainParam.goal= 0.1;
[net, tr] = train(net,P,t);
```

The problem runs, producing the following training record.

```
TRAINB, Epoch 0/100, MSE 0.5/0.1.
TRAINB, Epoch 25/100, MSE 0.181122/0.1.
TRAINB, Epoch 50/100, MSE 0.111233/0.1.
TRAINB, Epoch 64/100, MSE 0.0999066/0.1.
TRAINB, Performance goal met.
```

Thus, the performance goal is met in 64 epochs. The new weights and bias are

```
weights = net.iw{1,1}
weights =
   -0.0615   -0.2194
bias = net.b(1)
bias =
    [0.5899]
```

We can simulate the new network as shown below.

```
A = sim(net, p)
A =
    0.0282    0.9672    0.2741    0.4320,
```

We also can calculate the error.

```
err = t - sim(net,P)
err =
   -0.0282    0.0328   -0.2741    0.5680
```

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had we chosen a smaller error goal, but in this problem it is not possible to obtain a goal of 0. The network is limited

in its capability. See “Limitations and Cautions” at the end of this chapter for examples of various limitations.

This demonstration program `demolin2` shows the training of a linear neuron, and plots the weight trajectory and error during training

You also might try running the demonstration program `nnd10lc`. It addresses a classic and historically interesting problem, shows how a network can be trained to classify various patterns, and how the trained network responds when noisy patterns are presented.

Limitations and Cautions

Linear networks may only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate `lr` is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Since parabolas have only one minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have other various limitations. Some of them are discussed below.

Overdetermined Systems

Consider an overdetermined system. Suppose that we have a network to be trained with four 1-element input vectors and four targets. A perfect solution to $wp + b = t$ for each of the inputs may not exist, for there are four constraining equations and only one weight and one bias to adjust. However, the LMS rule will still minimize the error. You might try `demolin4` to see how this is done.

Underdetermined Systems

Consider a single linear neuron with one input. This time, in `demolin5`, we will train it on only one one-element input vector and its one-element target vector:

```
P = [+1.0];
T = [+0.5];
```

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can try `demoin5` to explore this topic.

Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ($S*R+S$ = number of weights and biases) as

constraints (Q = pairs of input/target vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with demonstration script `demolin6`, the network cannot solve the problem with zero error. You might want to try `demolin6`.

Too Large a Learning Rate

A linear network can always be trained with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Demonstration script `demolin7` shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by `maxlinlr`. The network is trained with two different learning rates to show the results of using too large a learning rate.

Summary

Single-layer linear networks can perform linear function approximation or pattern association.

Single-layer linear networks can be designed directly or trained with the Widrow-Hoff rule to find a minimum error solution. In addition, linear networks can be trained adaptively allowing the network to track changes in its environment.

The design of a single-layer linear network is constrained completely by the problem to be solved. The number of network inputs and the number of neurons in the layer are determined by the number of inputs and outputs required by the problem.

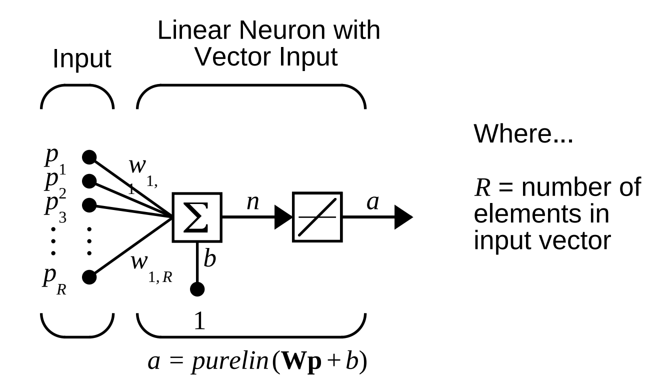
Multiple layers in a linear network do not result in a more powerful network, so the single layer is not a limitation. However, linear networks can solve only linear problems.

Nonlinear relationships between inputs and targets cannot be represented exactly by a linear network. The networks discussed in this chapter make a linear approximation with the minimum sum-squared error.

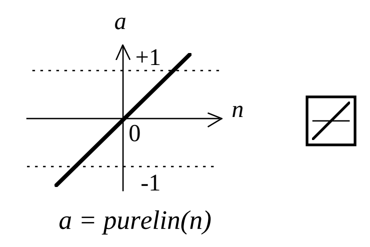
If the relationship between inputs and targets is linear or a linear approximation is desired, then linear networks are made for the job. Otherwise, backpropagation may be a good alternative.

Figures and Equations

Linear Neuron

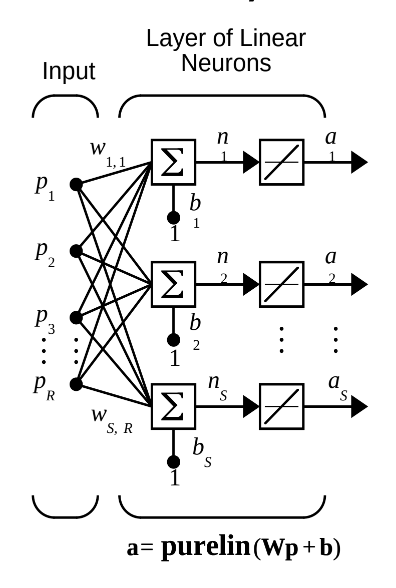


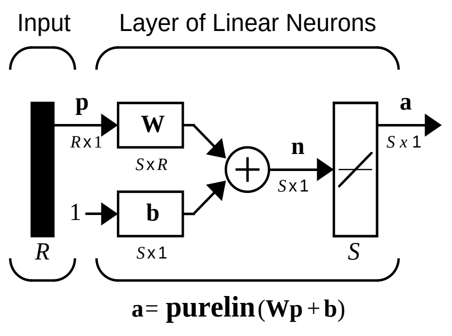
Purelin Transfer Function



Linear Transfer Function

Linear Network Layer



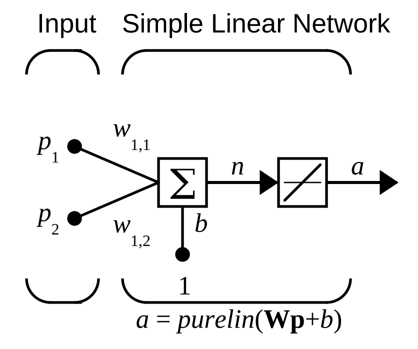


Where...

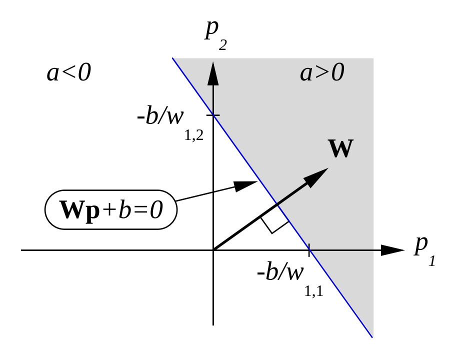
R = numberof elements in input vector

S = numberof neurons in layer

Simple Linear Network

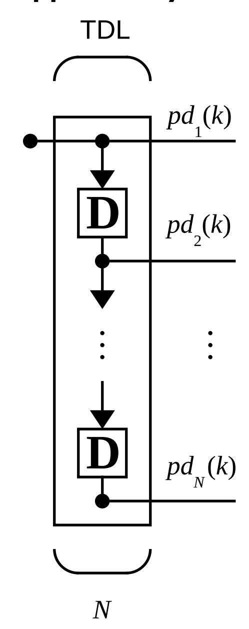


Decision Boundary

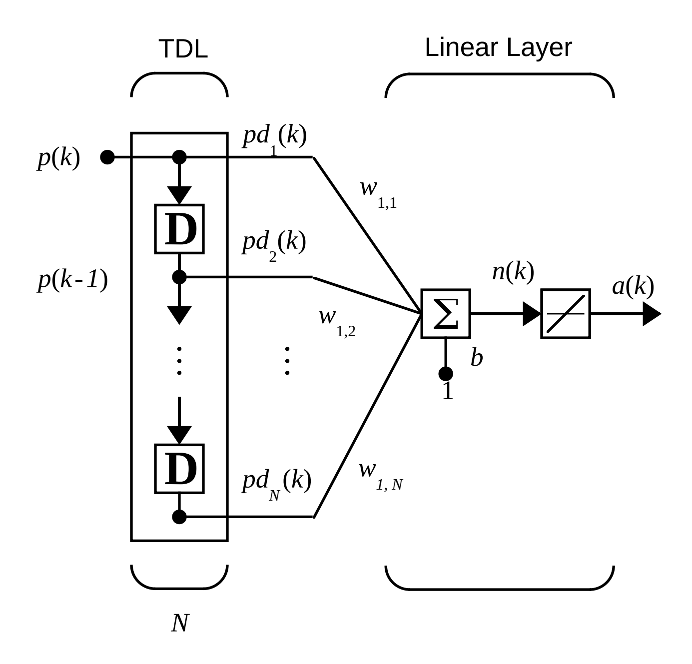


Mean Square Error

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$


Tapped Delay Line
TDL
D
D
pd1(k)
pd2(k)
pdN(k)
N

Linear Filter



LMS (Widrow-Hoff) Algorithm

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k) .$$

New Functions

This chapter introduces the following new functions.

Function	Description
`newlin`	Creates a linear layer.
`newlind`	Design a linear layer.

Function	Description
learnwh	Widrow-Hoff weight/bias learning rule.
purelin	A linear transfer function.

5

Backpropagation

Overview 5-2

Fundamentals 5-4
Architecture 5-4
Simulation (sim) 5-8
Training 5-8

Faster Training 5-14
Variable Learning Rate (traingda, traingdx) 5-14
Resilient Backpropagation (trainrp) 5-16
Conjugate Gradient Algorithms 5-17
Line Search Routines 5-23
Quasi-Newton Algorithms 5-26
Levenberg-Marquardt (trainlm) 5-28
Reduced Memory Levenberg-Marquardt (trainlm) 5-30

Speed and Memory Comparison 5-32
Summary 5-49

Improving Generalization 5-51
Regularization 5-52
Early Stopping 5-55
Summary and Discussion 5-57

Preprocessing and Postprocessing 5-61
Min and Max (premnmx, postmnmx, tramnmx) 5-61
Mean and Stand. Dev. (prestd, poststd, trastd) 5-62
Principal Component Analysis (prepca, trapca) 5-63
Post-Training Analysis (postreg) 5-64

Sample Training Session 5-66

Limitations and Cautions 5-71

Overview

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term *backpropagation* refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods. The Neural Network Toolbox implements a number of these variations. This chapter explains how to use each of these routines and discusses the advantages and disadvantages of each.

Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. There are two features of the Neural Network Toolbox that are designed to improve network generalization - regularization and early stopping. These features and their use are discussed later in this chapter.

This chapter also discusses preprocessing and postprocessing techniques, which can improve the efficiency of network training.

Before beginning this chapter you may want to read a basic reference on backpropagation, such as D.E Rumelhart, G.E. Hinton, R.J. Williams, "Learning internal representations by error propagation," D. Rumelhart and J. McClelland, editors. *Parallel Data Processing*, Vol.1, Chapter 8, the M.I.T. Press, Cambridge, MA 1986 pp. 318-362. This subject is also covered in detail in Chapters 11 and 12 of M.T. Hagan, H.B. Demuth, M.H. Beale, *Neural Network Design*, PWS Publishing Company, Boston, MA 1996.

The primary objective of this chapter is to explain how to use the backpropagation training functions in the toolbox to train feedforward neural networks to solve specific problems. There are generally four steps in the training process:

1. Assemble the training data
2. Create the network object
3. Train the network
4. Simulate the network response to new inputs

This chapter discusses a number of different training functions, but in using each function we generally follow these four steps.

The next section describes the basic feedforward network structure and demonstrates how to create a feedforward network object. Then the simulation and training of the network objects are presented.

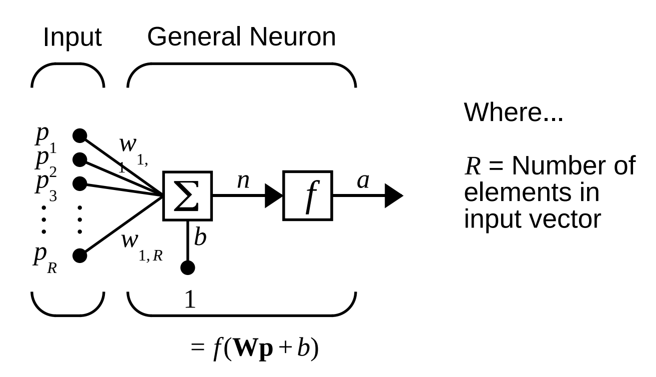
Fundamentals

Architecture

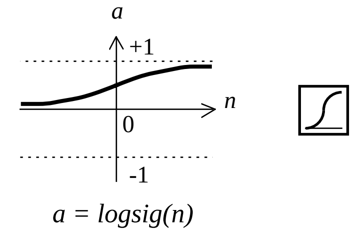
This section presents the architecture of the network that is most commonly used with the backpropagation algorithm - the multilayer feedforward network. The routines in the Neural Network Toolbox can be used to train more general networks; some of these will be briefly discussed in later chapters.

Neuron Model (tansig, logsig, purelin)

An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function f. Neurons may use any differentiable transfer function f to generate their output.



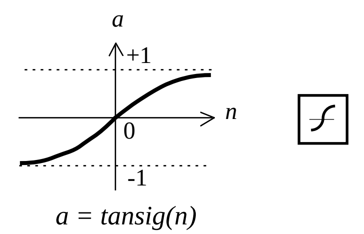
Multilayer networks often use the log-sigmoid transfer function `logsig`.



Log-Sigmoid Transfer Function

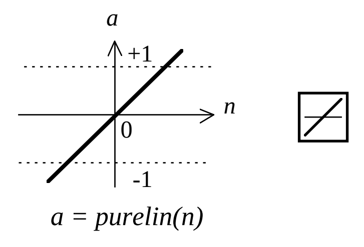
The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks may use the tan-sigmoid transfer function `tansig`.



Tan-Sigmoid Transfer Function

Occasionally, the linear transfer function `purelin` is used in backpropagation networks.



Linear Transfer Function

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

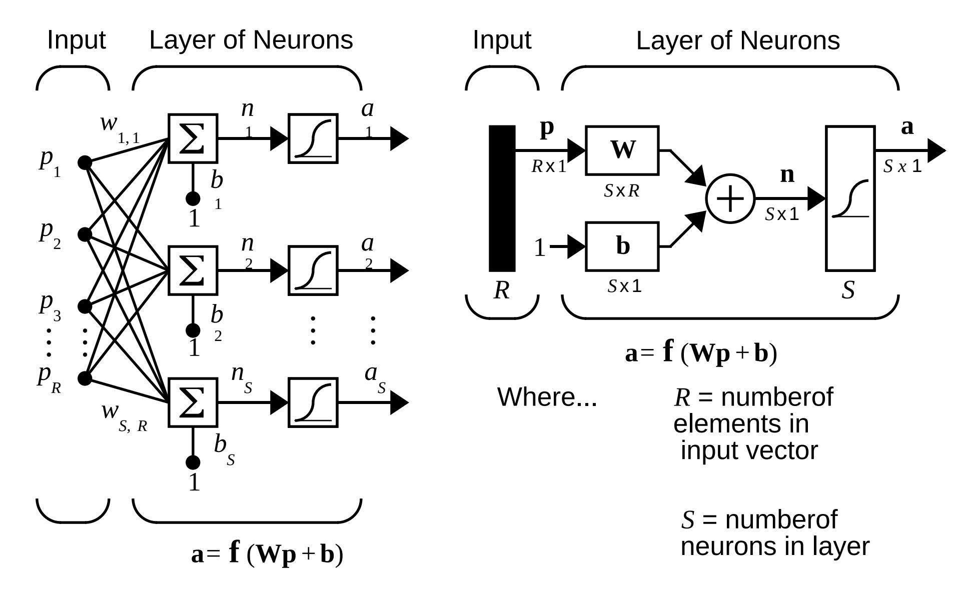
In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, `tansig`, `logsig`, and `purelin`, have a corresponding derivative function: `dtansig`, `dlogsig`, and `dpurelin`. To get the name of a transfer function's associated derivative function, call the transfer function with the string `'deriv'`.

```
tansig('deriv')
ans = dtansig
```

The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired. See “Advanced Topics” in Chapter 12.”

Feedforward Network

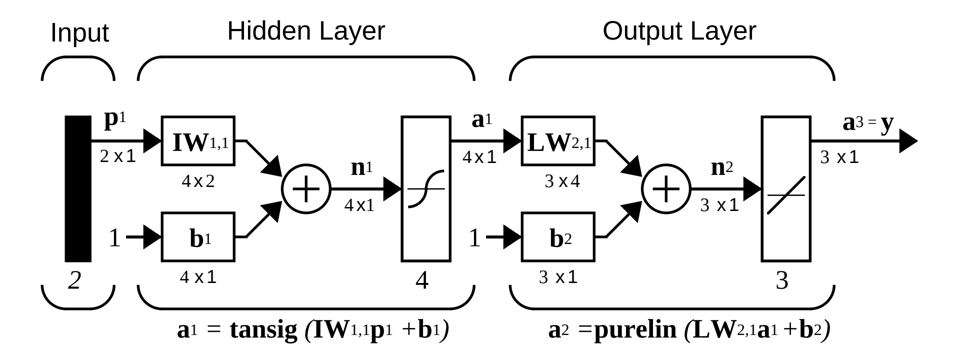
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range –1 to +1.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as `logsig`).

As noted in Chapter 2, for multiple-layer networks we use the number of the layers to determine the superscript on the weight matrices. The appropriate notation is used in the two-layer `tansig`/`purelin` network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities, arbitrarily well, given sufficient neurons in the hidden layer.

Creating a Network (newff). The first step in training a feedforward network is to create the network object. The function `newff` creates a feedforward network. It requires four inputs and returns the network object. The first input is an R by 2 matrix of minimum and maximum values for each of the R elements of the input vector. The second input is an array containing the sizes of each layer. The third input is a cell array containing the names of the transfer functions to be used in each layer. The final input contains the name of the training function to be used.

For example, the following command creates a two-layer network. There is one input vector with two elements. The values for the first element of the input vector range between -1 and 2, the values of the second element of the input vector range between 0 and 5. There are three neurons in the first layer and one neuron in the second (output) layer. The transfer function in the first layer is tan-sigmoid, and the output layer transfer function is linear. The training function is `traingd` (which is described in a later section).

```
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingd');
```

This command creates the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you may want to reinitialize the weights, or to perform a custom initialization. The next section explains the details of the initialization process.

Initializing Weights (init). Before training a feedforward network, the weights and biases must be initialized. The `newff` command will automatically initialize the

weights, but you may want to reinitialize them. This can be done with the command `init`. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized):

```
net = init(net);
```

For specifics on how the weights are initialized, see Chapter 12.

Simulation (sim)

The function `sim` simulates a network. `sim` takes the network input `p`, and the network object `net`, and returns the network outputs `a`. Here is how you can use `sim` to simulate the network we created above for a single input vector:

```
p = [1;2];
a = sim(net,p)
a =
   -0.1011
```

(If you try these commands, your output may be different, depending on the state of your random number generator when the network was initialized.) Below, `sim` is called to calculate the outputs for a concurrent set of three input vectors. This is the batch mode form of simulation, in which all of the input vectors are place in one matrix. This is much more efficient than presenting the vectors one at a time.

```
p = [1 3 2;2 4 1];
a=sim(net,p)
a =
  -0.1011   -0.2308    0.4955
```

Training

Once the network weights and biases have been initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior - network inputs `p` and target outputs `t`. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function `net.performFcn`. The default performance function for feedforward

networks is mean square error `mse` - the average squared error between the network outputs `a` and the target outputs `t`.

The remainder of this chapter describes several different training algorithms for feedforward networks. All of these algorithms use the gradient of the performance function to determine how to adjust the weights to minimize performance. The gradient is determined using a technique called backpropagation, which involves performing computations backwards through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapter 11 of [HDB96].

The basic backpropagation training algorithm, in which the weights are moved in the direction of the negative gradient, is described in the next section. Later sections describe more complex algorithms that increase the speed of convergence.

Backpropagation Algorithm

There are many variations of the backpropagation algorithm, several of which we discuss in this chapter. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient. One iteration of this algorithm can be written

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where $\mathbf{x}_k$ is a vector of current weights and biases, $\mathbf{g}_k$ is the current gradient, and α_k is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In the incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In the batch mode all of the inputs are applied to the network before the weights are updated. The next section describes the batch mode of training; incremental training will be discussed in a later chapter.

Batch Training (train). In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases. For a discussion of batch training with the backpropagation algorithm see page 12-7 of [HDB96].

Batch Gradient Descent (traingd). The batch steepest descent training function is traingd. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network trainFcn to traingd, and then call the function train. There is only one training function associated with a given network.

There are seven training parameters associated with traingd: epochs, show, goal, time, min_grad, max_fail, and lr. The learning rate lr is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. See page 12-8 of [HDB96] for a discussion of the choice of learning rate.

The training status is displayed for every show iteration of the algorithm. (If show is set to NaN, then the training status never displays.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds epochs, if the performance function drops below goal, if the magnitude of the gradient is less than mingrad, or if the training time is longer than time seconds. We discuss max_fail, which is associated with the early stopping technique, in the section on improving generalization.

The following code creates a training set of inputs p and targets t. For batch training, all of the input vectors are placed in one matrix.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
```

Next, we create the feedforward network. Here we use the function minmax to determine the range of the inputs to be used in creating the network.

```
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
```

At this point, we might want to modify some of the default training parameters.

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

If you want to use the default training parameters, the above commands are not necessary.

Now we are ready to train the network.

```
[net,tr]=train(net,p,t);
   TRAINGD, Epoch 0/300, MSE 1.59423/1e-05, Gradient 2.76799/
1e-10
   TRAINGD, Epoch 50/300, MSE 0.00236382/1e-05, Gradient
0.0495292/1e-10
   TRAINGD, Epoch 100/300, MSE 0.000435947/1e-05, Gradient
0.0161202/1e-10
   TRAINGD, Epoch 150/300, MSE 8.68462e-05/1e-05, Gradient
0.00769588/1e-10
   TRAINGD, Epoch 200/300, MSE 1.45042e-05/1e-05, Gradient
0.00325667/1e-10
   TRAINGD, Epoch 211/300, MSE 9.64816e-06/1e-05, Gradient
0.00266775/1e-10
   TRAINGD, Performance goal met.
```

The training record `tr` contains information about the progress of training. An example of its use is given in the Sample Training Session near the end of this chapter.

Now the trained network can be simulated to obtain its response to the inputs in the training set.

```
a = sim(net,p)
a =
   -1.0010   -0.9989    1.0018    0.9985
```

Try the Neural Network Design Demonstration `nnd12sd1`[HDB96] for an illustration of the performance of the batch gradient descent algorithm.

Batch Gradient Descent with Momentum (traingdm). In addition to `traingd`, there is another batch algorithm for feedforward networks that often provides faster convergence - `traingdm`, steepest descent with momentum. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low-pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12-9 of [HDB96] for a discussion of momentum.

Momentum can be added to backpropagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a momentum constant, `mc`, which can be any number between 0 and 1. When the momentum constant is 0, a weight change is based solely on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored. The gradient is computed by summing the gradients calculated at each training example, and the weights and biases are only updated after all training examples have been presented.

If the new performance function on a given iteration exceeds the performance function on a previous iteration by more than a predefined ratio `max_perf_inc` (typically 1.04), the new weights and biases are discarded, and the momentum coefficient `mc` is set to zero.

The batch form of gradient descent with momentum is invoked using the training function `traingdm`. The `traingdm` function is invoked using the same steps shown above for the `traingd` function, except that the `mc`, `lr` and `max_perf_inc` learning parameters can all be set.

In the following code we recreate our previous network and retrain it using gradient descent with momentum. The training parameters for `traingdm` are the same as those for `traingd`, with the addition of the momentum factor `mc` and the maximum performance increase `max_perf_inc`. (The training parameters are reset to the default values whenever `net.trainFcn` is set to `traingdm`.)

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINGDM, Epoch 0/300, MSE 3.6913/1e-05, Gradient 4.54729/
1e-10
   TRAINGDM, Epoch 50/300, MSE 0.00532188/1e-05, Gradient
0.213222/1e-10
```

```
   TRAINGDM, Epoch 100/300, MSE 6.34868e-05/1e-05, Gradient 
0.0409749/1e-10
   TRAINGDM, Epoch 114/300, MSE 9.06235e-06/1e-05, Gradient 
0.00908756/1e-10
   TRAINGDM, Performance goal met.
a = sim(net,p)
a =
    -1.0026   -1.0044    0.9969    0.9992
```

Note that since we reinitialized the weights and biases before training (by calling `newff` again), we obtain a different mean square error than we did using `traingd`. If we were to reinitialize and train again using `traingdm`, we would get yet a different mean square error. The random choice of initial weights and biases will affect the performance of the algorithm. If you want to compare the performance of different algorithms, you should test each using several different sets of initial weights and biases. You may want to use `net=init(net)` to reinitialize the weights, rather than recreating the entire network with `newff`.

Try the Neural Network Design Demonstration `nnd12mo` [HDB96] for an illustration of the performance of the batch momentum algorithm.

Faster Training

The previous section presented two backpropagation training algorithms: gradient descent, and gradient descent with momentum. These two methods are often too slow for practical problems. In this section we discuss several high performance algorithms that can converge from ten to one hundred times faster than the algorithms discussed previously. All of the algorithms in this section operate in the batch mode and are invoked using `train`.

These faster algorithms fall into two main categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. One heuristic modification is the momentum technique, which was presented in the previous section. This section discusses two more heuristic techniques: variable learning rate backpropagation, `traingda`; and resilient backpropagation `trainrp`.

The second category of fast algorithms uses standard numerical optimization techniques. (See Chapter 9 of [HDB96] for a review of basic numerical optimization.) Later in this section we present three types of numerical optimization techniques for neural network training: conjugate gradient (`traincgf`, `traincgp`, `traincgb`, `trainscg`), quasi-Newton (`trainbfg`, `trainoss`), and Levenberg-Marquardt (`trainlm`).

Variable Learning Rate (traingda, traingdx)

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm may oscillate and become unstable. If the learning rate is too small, the algorithm will take too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by `traingd`. First, the initial network output and error are calculated. At

each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio `max_perf_inc` (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by `lr_dec` = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by `lr_inc` = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it gets decreased until stable learning resumes.

Try the Neural Network Design Demonstration `nnd12vl` [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function `traingda`, which is called just like `traingd`, except for the additional training parameters `max_perf_inc`, `lr_dec`, and `lr_inc`. Here is how it is called to train our previous two-layer network:

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingda');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINGDA, Epoch 0/300, MSE 1.71149/1e-05, Gradient 2.6397/
1e-06
   TRAINGDA, Epoch 44/300, MSE 7.47952e-06/1e-05, Gradient
0.00251265/1e-06
   TRAINGDA, Performance goal met.
a = sim(net,p)
a =
   -1.0036   -0.9960    1.0008    0.9991
```

The function traingdx combines adaptive learning rate with momentum training. It is invoked in the same way as traingda, except that it has the momentum coefficient mc as an additional training parameter.

Resilient Backpropagation (trainrp)

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called “squashing” functions, since they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slope must approach zero as the input gets large. This causes a problem when using steepest descent to train a multilayer network with sigmoid functions, since the gradient can have a very small magnitude; and therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased. A complete description of the Rprop algorithm is given in [ReBr93].

In the following code we recreate our previous network and train it using the Rprop algorithm. The training parameters for trainrp are epochs, show, goal, time, min_grad, max_fail, delt_inc, delt_dec, delta0, deltamax. We have previously discussed the first eight parameters. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, we leave most of the training parameters at the default values. We do reduce show below our previous value, because Rprop generally converges much faster than the previous algorithms.

```
p = [-1 -1 2 2;0 5 0 5];
```

```
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainrp');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINRP, Epoch 0/300, MSE 0.469151/1e-05, Gradient 1.4258/
1e-06
   TRAINRP, Epoch 10/300, MSE 0.000789506/1e-05, Gradient
0.0554529/1e-06
   TRAINRP, Epoch 20/300, MSE 7.13065e-06/1e-05, Gradient
0.00346986/1e-06
   TRAINRP, Performance goal met.
a = sim(net,p)
a =
   -1.0026   -0.9963    0.9978    1.0017
```

Rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. We do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

Conjugate Gradient Algorithms

The basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In this section, we present four different variations of conjugate gradient algorithms.

See page 12-14 of [HDB96] for a discussion of conjugate gradient algorithms and their application to neural networks.

In most of the training algorithms that we discussed up to this point, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size, which minimizes the performance function along that line. There are five

different search functions included in the toolbox, and these are discussed at the end of this section. Any of these search functions can be used interchangeably with a variety of the training functions described in the remainder of this chapter. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application. An appropriate default search function is assigned to each training function, but this can be modified by the user.

Fletcher-Reeves Update (traincgf)

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of conjugate gradient are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

In the following code, we reinitialize our previous network and retrain it using the Fletcher-Reeves version of the conjugate gradient algorithm. The training

parameters for traincgf are epochs, show, goal, time, min_grad, max_fail, srchFcn, scal_tol, alpha, beta, delta, gama, low_lim, up_lim, maxstep, minstep, bmax. We have previously discussed the first six parameters. The parameter srchFcn is the name of the line search function. It can be any of the functions described later in this section (or a user-supplied function). The remaining parameters are associated with specific line search routines and are described later in this section. The default line search routine srchcha is used in this example. traincgf generally converges in fewer iterations than trainrp (although there is more computation required in each iteration).

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGF-srchcha, Epoch 0/300, MSE 2.15911/1e-05, Gradient
3.17681/1e-06
   TRAINCGF-srchcha, Epoch 5/300, MSE 0.111081/1e-05, Gradient
0.602109/1e-06
   TRAINCGF-srchcha, Epoch 10/300, MSE 0.0095015/1e-05, Gradient
0.197436/1e-06
   TRAINCGF-srchcha, Epoch 15/300, MSE 0.000508668/1e-05,
Gradient 0.0439273/1e-06
   TRAINCGF-srchcha, Epoch 17/300, MSE 1.33611e-06/1e-05,
Gradient 0.00562836/1e-06
   TRAINCGF, Performance goal met.
a = sim(net,p)
a =
   -1.0001   -1.0023    0.9999    1.0002
```

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than trainrp, although the results will vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms, so they are often a good choice for networks with a large number of weights.

Try the Neural Network Design Demonstration nnd12cg [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

Polak-Ribiére Update (traincgp)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

In the following code, we recreate our previous network and train it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for `traincgp` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epoch` are set to the same values as they were for `traincgf`.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgp');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGP-srchcha, Epoch 0/300, MSE 1.21966/1e-05, Gradient
1.77008/1e-06
   TRAINCGP-srchcha, Epoch 5/300, MSE 0.227447/1e-05, Gradient
0.86507/1e-06
   TRAINCGP-srchcha, Epoch 10/300, MSE 0.000237395/1e-05,
Gradient 0.0174276/1e-06
   TRAINCGP-srchcha, Epoch 15/300, MSE 9.28243e-05/1e-05,
Gradient 0.00485746/1e-06
   TRAINCGP-srchcha, Epoch 20/300, MSE 1.46146e-05/1e-05,
Gradient 0.000912838/1e-06
```

```
   TRAINCGP-srchcha, Epoch 25/300, MSE 1.05893e-05/1e-05,
Gradient 0.00238173/1e-06
   TRAINCGP-srchcha, Epoch 26/300, MSE 9.10561e-06/1e-05,
Gradient 0.00197441/1e-06
   TRAINCGP, Performance goal met.
a = sim(net,p)
a =
   -0.9967   -1.0018    0.9958    1.0022
```

The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

Powell-Beale Restarts (traincgb)

For all conjugate gradient algorithms, the search direction will be periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. For this technique we will restart if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality.

$$\left|\mathbf{g}_{k-1}^T \mathbf{g}_k\right| \geq 0.2\left\|\mathbf{g}_k\right\|^2$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

In the following code, we recreate our previous network and train it using the Powell-Beale version of the conjugate gradient algorithm. The training parameters for `traincgb` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epoch` are set to the same values as they were for `traincgf`.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgb');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

```
[net,tr]=train(net,p,t);
   TRAINCGB-srchcha, Epoch 0/300, MSE 2.5245/1e-05, Gradient 
3.66882/1e-06
   TRAINCGB-srchcha, Epoch 5/300, MSE 4.86255e-07/1e-05, Gradient 
0.00145878/1e-06
   TRAINCGB, Performance goal met.
a = sim(net,p)
a =
   -0.9997   -0.9998    1.0000    1.0014
```

The traincgb routine has performance that is somewhat better than traincgp for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

Scaled Conjugate Gradient (trainscg)

Each of the conjugate gradient algorithms that we have discussed so far requires a line search at each iteration. This line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller [Moll93], was designed to avoid the time-consuming line search. This algorithm is too complex to explain in a few lines, but the basic idea is to combine the model-trust region approach (used in the Levenberg-Marquardt algorithm described later), with the conjugate gradient approach. See {Moll93] for a detailed explanation of the algorithm.

In the following code, we reinitialize our previous network and retrain it using the scaled conjugate gradient algorithm. The training parameters for trainscg are epochs, show, goal, time, min_grad, max_fail, sigma, lambda. We have previously discussed the first six parameters. The parameter sigma determines the change in the weight for the second derivative approximation. The parameter lambda regulates the indefiniteness of the Hessian. The parameters show and epoch are set to 10 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainscg');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

```
   TRAINSCG, Epoch 0/300, MSE 4.17697/1e-05, Gradient 5.32455/
1e-06
   TRAINSCG, Epoch 10/300, MSE 2.09505e-05/1e-05, Gradient
0.00673703/1e-06
   TRAINSCG, Epoch 11/300, MSE 9.38923e-06/1e-05, Gradient
0.0049926/1e-06
   TRAINSCG, Performance goal met.
a = sim(net,p)
a =
   -1.0057   -1.0008    1.0019    1.0005
```

The `trainscg` routine may require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of Fletcher-Reeves.

Line Search Routines

Several of the conjugate gradient and quasi-Newton algorithms require that a line search be performed. In this section, we describe five different line searches you can use. To use any of these search routines, you simply set the training parameter `srchFcn` equal to the name of the desired search function, as described in previous sections. It is often difficult to predict which of these routines provide the best results for any given problem, but we set the default search function to an appropriate initial choice for each training function, so you never need to modify this parameter.

Golden Section Search (srchgol)

The golden section search `srchgol` is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determines a section of the interval that can be discarded, and a new interior point is placed within the new interval.

This procedure is continued until the interval of uncertainty is reduced to a width of `tol`, which is equal to `delta/scale_tol`.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the Neural Network Design Demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.

Brent's Search (srchbre)

Brent's search is a linear search, which is a hybrid combination of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate that is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods may take many iterations to become apparent. Brent's search attempts to combine the best features of both approaches.

For Brent's search we begin with the same interval of uncertainty that we used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.

See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm may require more performance evaluations than algorithms that use derivative information.

Hybrid Bisection-Cubic Search (srchhyb)

Like Brent's search, `srchhyb` is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty and the performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is

obtained by using the value of the performance and its derivative at the two end points. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent's algorithm.

Charalambous' Search (srchcha)

The method of Charalambous `srchcha` was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like the previous two methods, it is a hybrid search. It uses a cubic interpolation, together with a type of sectioning.

See [Char92] for a description of Charalambous' search. We have used this routine as the default search for most of the conjugate gradient algorithms, since it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with fewer steps. This is not true for all problems, and you may want to experiment with other line searches.

Backtracking (srchbac)

The backtracking search routine `srchbac` is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and at a step multiplier of 1. Also it uses the value of the derivative of performance at the current point, to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in [DeSc83]. It was used as the default line search for the quasi-Newton algorithms, although it may not be the best technique for all problems.

Quasi-Newton Algorithms

BFGS Algorithm (trainbgf)

Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where $\mathbf{A}_k$ is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm has been implemented in the `trainbfg` routine.

In the following code, we reinitialize our previous network and retrain it using the BFGS quasi-Newton algorithm. The training parameters for `trainbfg` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINBFG-srchbac, Epoch 0/300, MSE 0.492231/1e-05, Gradient
2.16307/1e-06
```

```
   TRAINBFG-srchbac, Epoch 5/300, MSE 0.000744953/1e-05, Gradient
0.0196826/1e-06
   TRAINBFG-srchbac, Epoch 8/300, MSE 7.69867e-06/1e-05, Gradient
0.00497404/1e-06
   TRAINBFG, Performance goal met.
a = sim(net,p)
a =
   -0.9995   -1.0004    1.0008    0.9945
```

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is $n \times n$, where n is equal to the number of weights and biases in the network. For very large networks it may be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

One Step Secant Algorithm (trainoss)

Since the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

In the following code, we reinitialize our previous network and retrain it using the one-step secant algorithm. The training parameters for `trainoss` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainoss');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

```
   TRAINOSS-srchbac, Epoch 0/300, MSE 0.665136/1e-05, Gradient
1.61966/1e-06
   TRAINOSS-srchbac, Epoch 5/300, MSE 0.000321921/1e-05, Gradient
0.0261425/1e-06
   TRAINOSS-srchbac, Epoch 7/300, MSE 7.85697e-06/1e-05, Gradient
0.00527342/1e-06
   TRAINOSS, Performance goal met.
a = sim(net,p)
a =
   -1.0035   -0.9958    1.0014    0.9997
```

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

Levenberg-Marquardt (trainlm)

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $\mathbf{e}$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift towards Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

In the following code, we reinitialize our previous network and retrain it using the Levenberg-Marquardt algorithm. The training parameters for `trainlm` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `mu`, `mu_dec`, `mu_inc`, `mu_max`, `mem_reduc`. We have discussed the first six parameters earlier. The parameter `mu` is the initial value for μ. This value is multiplied by `mu_dec` whenever the performance function is reduced by a step. It is multiplied by `mu_inc` whenever a step would increase the performance function. If `mu` becomes larger than `mu_max`, the algorithm is stopped. The parameter `mem_reduc` is used to control the amount of memory used by the algorithm. It is discussed in the next section. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINLM, Epoch 0/300, MSE 2.7808/1e-05, Gradient 7.77931/1e-10
   TRAINLM, Epoch 4/300, MSE 3.67935e-08/1e-05, Gradient
0.000808272/1e-10
   TRAINLM, Performance goal met.
a = sim(net,p)
a =
   -1.0000   -1.0000    1.0000    0.9996
```

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has a very efficient MATLAB implementation, since the solution of the matrix

equation is a built-in function, so its attributes become even more pronounced in a MATLAB setting.

Try the Neural Network Design Demonstration nnd12m [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

Reduced Memory Levenberg-Marquardt (trainlm)

The main drawback of the Levenberg-Marquardt algorithm is that it requires the storage of some matrices that can be quite large for certain problems. The size of the Jacobian matrix is $Q \times n$, where Q is the number of training sets and n is the number of weights and biases in the network. It turns out that this matrix does not have to be computed and stored as a whole. For example, if we were to divide the Jacobian into two equal submatrices we could compute the approximate Hessian matrix as follows:

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} = \begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{J}_1 \\ \mathbf{J}_2 \end{bmatrix} = \mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{J}_2$$

Therefore, the full Jacobian does not have to exist at one time. The approximate Hessian can be computed by summing a series of subterms. Once one subterm has been computed, the corresponding submatrix of the Jacobian can be cleared.

When using the training function `trainlm`, the parameter `mem_reduc` is used to determine how many rows of the Jacobian are to be computed in each submatrix. If `mem_reduc` is set to 1, then the full Jacobian is computed, and no memory reduction is achieved. If `mem_reduc` is set to 2, then only half of the Jacobian will be computed at one time. This saves half of the memory used by the calculation of the full Jacobian.

There is a drawback to using memory reduction. A significant computational overhead is associated with computing the Jacobian in submatrices. If you have enough memory available, then it is better to set `mem_reduc` to 1 and to compute the full Jacobian. If you have a large training set, and you are running out of memory, then you should set `mem_reduc` to 2, and try again. If you still run out of memory, continue to increase `mem_reduc`.

Even if you use memory reduction, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions $n \times n$. If your network is very large, then you may run out of memory. If this is the

case, then you will want to try `trainscg`, `trainrp`, or one of the conjugate gradient algorithms.

Speed and Memory Comparison

It is very difficult to know which training algorithm will be the fastest for a given problem. It will depend on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). In this section we perform a number of benchmark comparisons of the various training algorithms. We train feedforward networks on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple “toy” problems, while the other four are “real world” problems. We use networks with a variety of different architectures and complexities, and we train the networks to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms we use to identify them.

Acronym	Algorithm
LM	`trainlm` - Levenberg-Marquardt
BFG	`trainbfg` - BFGS Quasi-Newton
RP	`trainrp` - Resilient Backpropagation
SCG	`trainscg` - Scaled Conjugate Gradient
CGB	`traincgb` - Conjugate Gradient with Powell/Beale Restarts
CGF	`traincgf` - Fletcher-Powell Conjugate Gradient
CGP	`traincgp` - Polak-Ribiére Conjugate Gradient
OSS	`trainoss` - One-Step Secant
GDX	`traingdx` - Variable Learning Rate Backpropagation

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

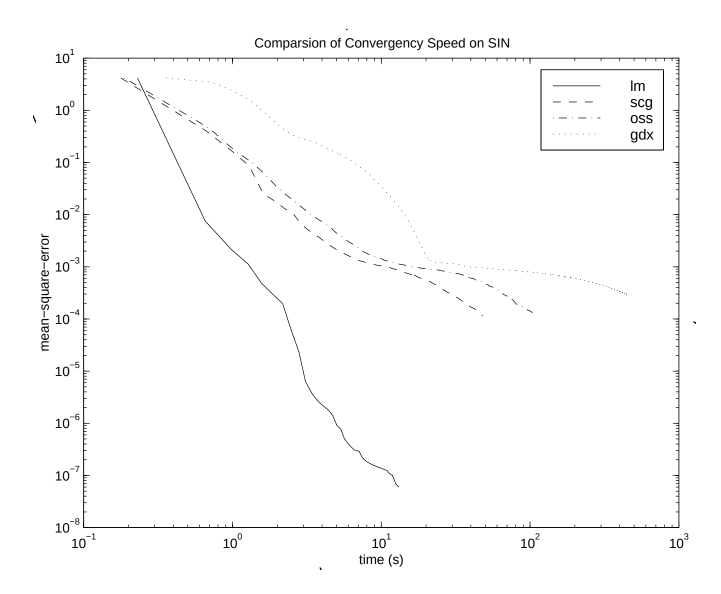
Problem Title	Problem Type	Network Structure	Error Goal	Computer
SIN	Function Approx.	1-5-1	0.002	Sun Sparc 2
PARITY	Pattern Recog.	3-10-10-1	0.001	Sun Sparc 2
ENGINE	Function Approx.	2-30-2	0.005	Sun Enterprise 4000
CANCER	Pattern Recog.	9-5-5-2	0.012	Sun Sparc 2
CHOLESTEROL	Function Approx.	21-15-3	0.027	Sun Sparc 20
DIABETES	Pattern Recog.	8-15-15-2	0.05	Sun Sparc 20

SIN Data Set

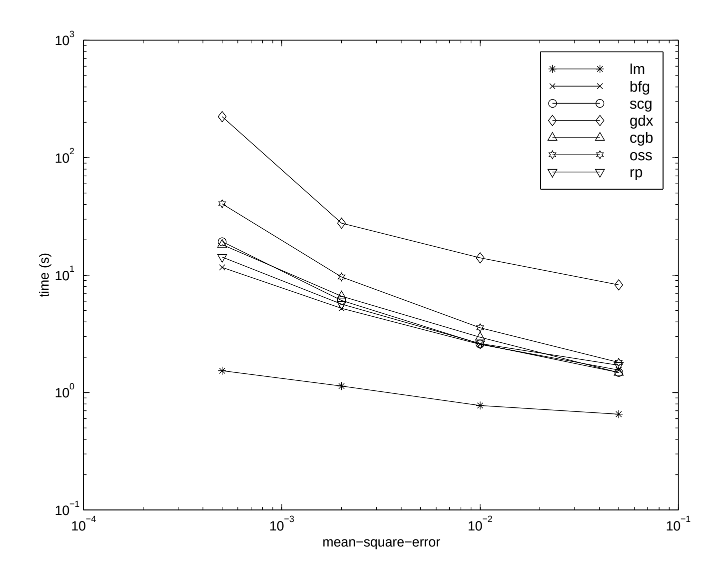
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with tansig transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training the network using nine different training algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.002. The fastest algorithm for this problem is the Levenberg-Marquardt algorithm. On the average, it is over four times faster than the next fastest algorithm. This is the type of problem for which the LM algorithm is best suited — a function approximation problem where the network has less than one hundred weights and the approximation must be very accurate.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	1.14	1.00	0.65	1.83	0.38
BFG	5.22	4.58	3.17	14.38	2.08
RP	5.67	4.97	2.66	17.24	3.72
SCG	6.09	5.34	3.18	23.64	3.81
CGB	6.61	5.80	2.99	23.65	3.67
CGF	7.86	6.89	3.57	31.23	4.76
CGP	8.24	7.23	4.07	32.32	5.03
OSS	9.64	8.46	3.97	59.63	9.79
GDX	27.69	24.29	17.21	258.15	43.65

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is demonstrated in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here we can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Here, we can see that as the error goal is reduced, the improvement provided by the LM algorithm becomes more pronounced. Some algorithms perform better as the error goal is reduced (LM and BFG), and other algorithms degrade as the error goal is reduced (OSS and GDX).

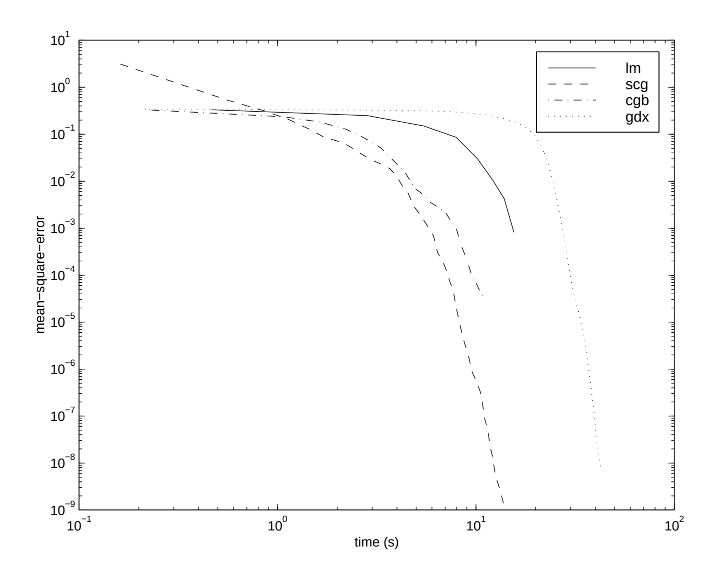


PARITY Data Set

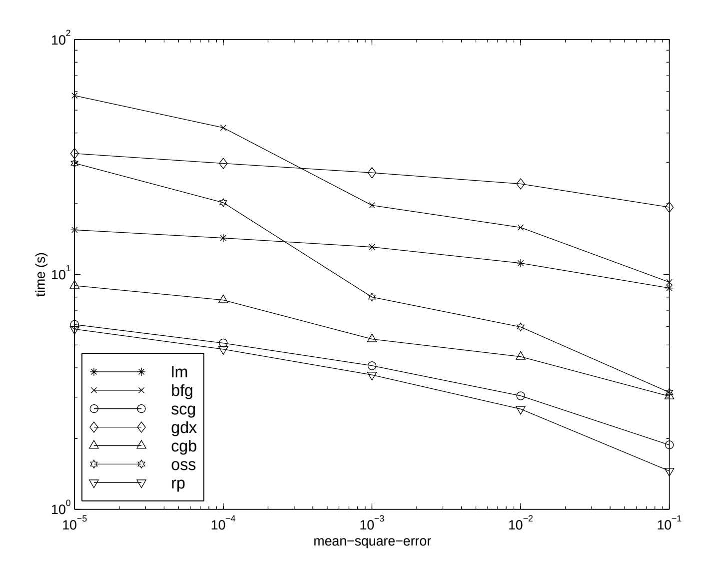
The second benchmark problem is a simple pattern recognition problem — detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a one; otherwise, it should output a minus one. The network used for this problem is a 3-10-10-1 network with tansig neurons in each layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Since the output neurons in pattern recognition problems will generally be saturated, we will not be operating in the linear region.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	3.73	1.00	2.35	6.89	1.26
SCG	4.09	1.10	2.36	7.48	1.56
CGP	5.13	1.38	3.50	8.73	1.05
CGB	5.30	1.42	3.91	11.59	1.35
CGF	6.62	1.77	3.96	28.05	4.32
OSS	8.00	2.14	5.06	14.41	1.92
LM	13.07	3.50	6.48	23.78	4.96
BFG	19.68	5.28	14.19	26.64	2.85
GDX	27.07	7.26	25.21	28.52	0.86

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is demonstrated in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (OSS and BFG).

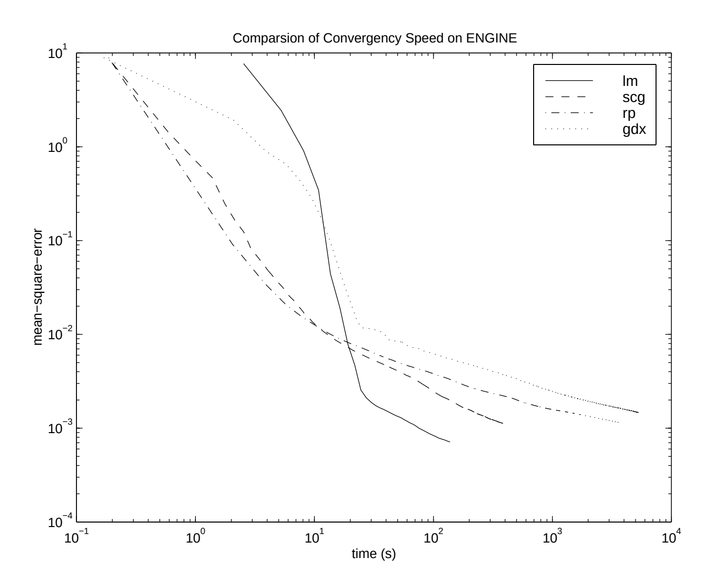


ENGINE Data Set

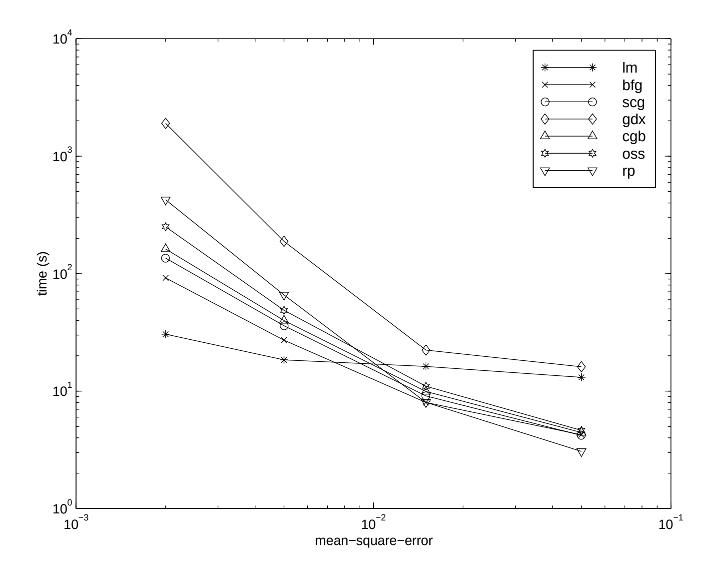
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tansig neurons in the hidden layer and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus. 16), and the advantages of the LM algorithm decrease as the number of network parameters increase.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	18.45	1.00	12.01	30.03	4.27
BFG	27.12	1.47	16.42	47.36	5.95
SCG	36.02	1.95	19.39	52.45	7.78
CGF	37.93	2.06	18.89	50.34	6.12
CGB	39.93	2.16	23.33	55.42	7.50
CGP	44.30	2.40	24.99	71.55	9.89
OSS	48.71	2.64	23.51	80.90	12.33
RP	65.91	3.57	31.83	134.31	34.24
GDX	188.50	10.22	81.59	279.90	66.67

The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (GDX and RP), while the LM algorithm improves.



CANCER Data Set

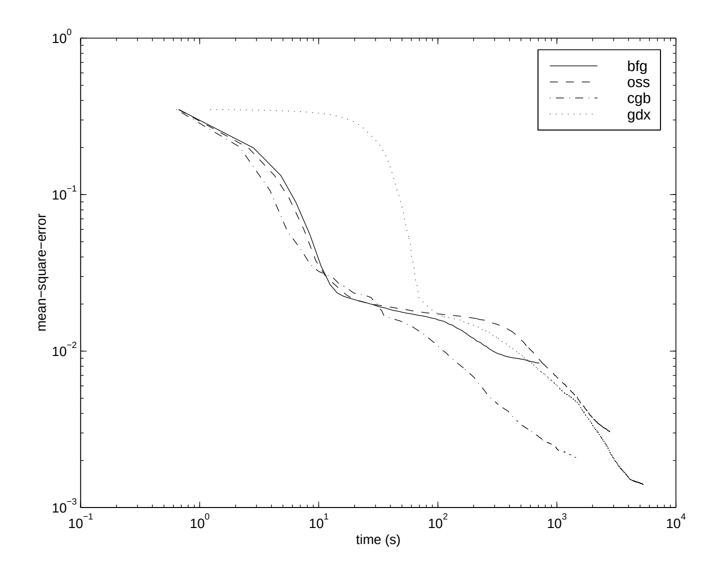
The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. The data was obtained from the University of Wisconsin Hospitals, Madison, from Dr. William H. Wolberg. The network used for this problem is a 9-5-5-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As we mentioned with the parity data set, the LM algorithm does not perform as well

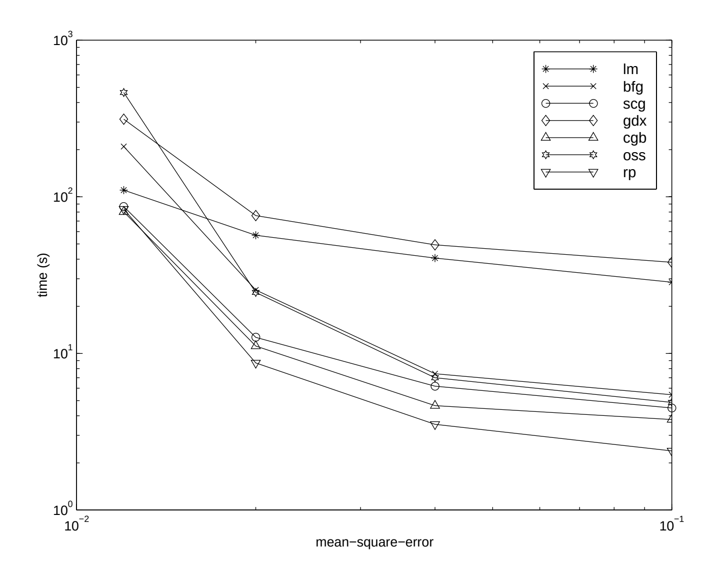
on pattern recognition problems as it does on function approximation problems.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
CGB	80.27	1.00	55.07	102.31	13.17
RP	83.41	1.04	59.51	109.39	13.44
SCG	86.58	1.08	41.21	112.19	18.25
CGP	87.70	1.09	56.35	116.37	18.03
CGF	110.05	1.37	63.33	171.53	30.13
LM	110.33	1.37	58.94	201.07	38.20
BFG	209.60	2.61	118.92	318.18	58.44
GDX	313.22	3.90	166.48	446.43	75.44
OSS	463.87	5.78	250.62	599.99	97.35

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem we don't see as much variation in performance as we have seen in previous problems.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (OSS and BFG) while the LM algorithm improves. It is typical of the LM algorithm on any problem that its performance improves relative to other algorithms as the error goal is reduced.



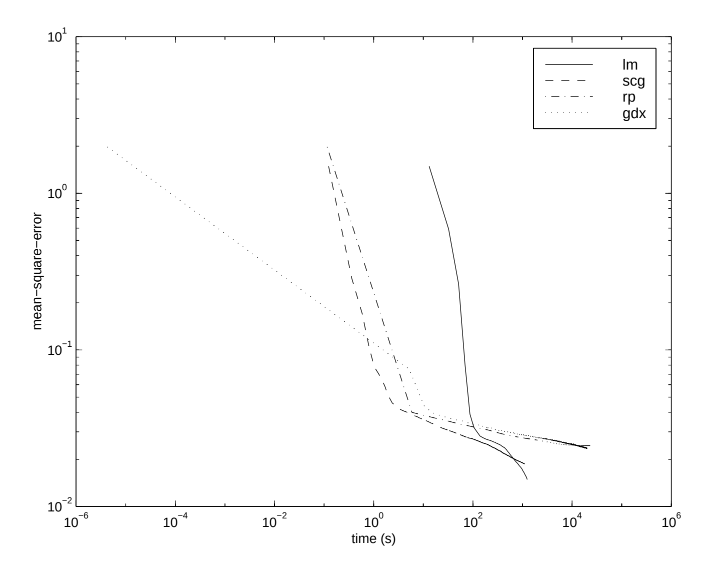
CHOLESTEROL Data Set

The fifth benchmark problem is a realistic function approximation (or nonlinear regression) problem. The objective of the network is to predict cholesterol levels (ldl, hdl and vldl) based on measurements of 21 spectral components. The data was obtained from Dr. Neil Purdie, Department of Chemistry, Oklahoma State University [PuLu92]. The network used for this problem is a 21-15-3 network with tansig neurons in the hidden layers and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 20 different trials (10 trials for RP and GDX), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.027.

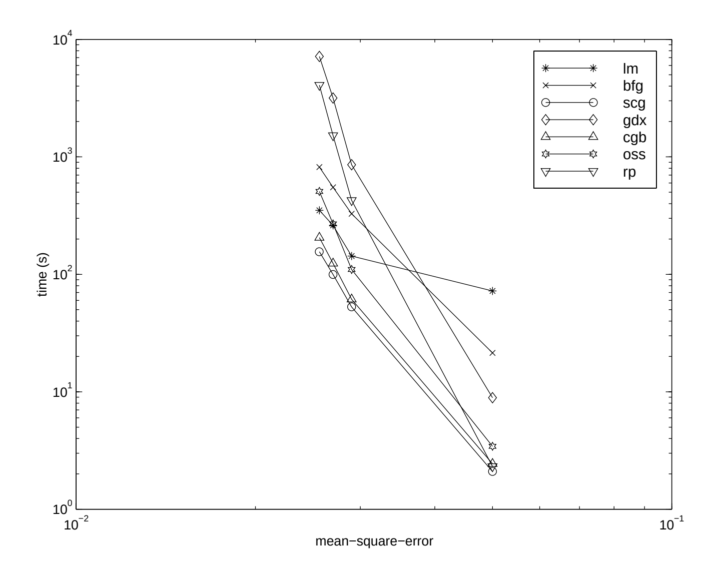
The scaled conjugate gradient algorithm has the best performance on this problem, although all of the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases in the network has increased again (378 versus 152 versus 16). As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
SCG	99.73	1.00	83.10	113.40	9.93
CGP	121.54	1.22	101.76	162.49	16.34
CGB	124.06	1.24	107.64	146.90	14.62
CGF	136.04	1.36	106.46	167.28	17.67
LM	261.50	2.62	103.52	398.45	102.06
OSS	268.55	2.69	197.84	372.99	56.79
BFG	550.92	5.52	471.61	676.39	46.59
RP	1519.00	15.23	581.17	2256.10	557.34
GDX	3169.50	31.78	2514.90	4168.20	610.52

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, we can see that the LM algorithm is able to drive the mean square error to a lower level than the other algorithms. The SCG and RP algorithms provide the fastest initial convergence.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. We can see that the LM and BFG algorithms improve relative to the other algorithms as the error goal is reduced.



DIABETES Data Set

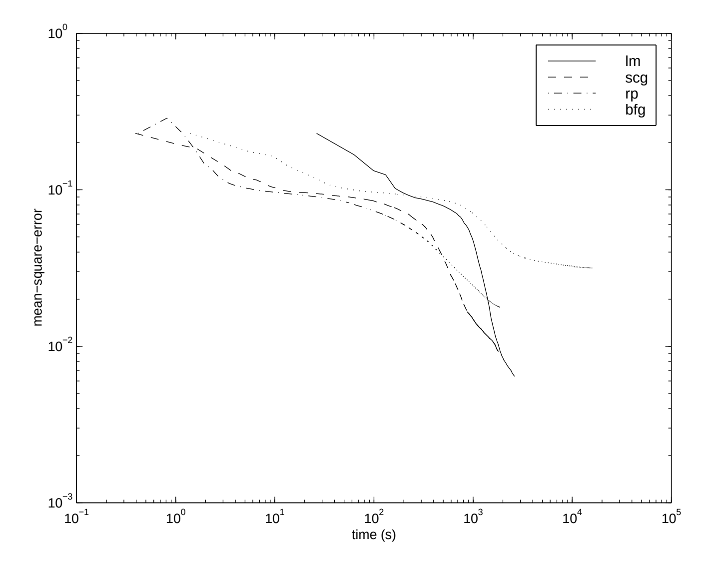
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide if an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance test, etc.). The data was obtained from the University of California, Irvine, machine learning data base. The network used for this problem is an 8-15-15-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 10 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.05.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems we have considered. The RP algorithm works well on all of the pattern recognition problems. This is reasonable, since that algorithm was designed to overcome the difficulties caused by training with sigmoid functions, which have very small slopes when operating far from the center point. For pattern recognition problems, we use sigmoid transfer functions in the output layer, and we want the network to operate at the tails of the sigmoid function.

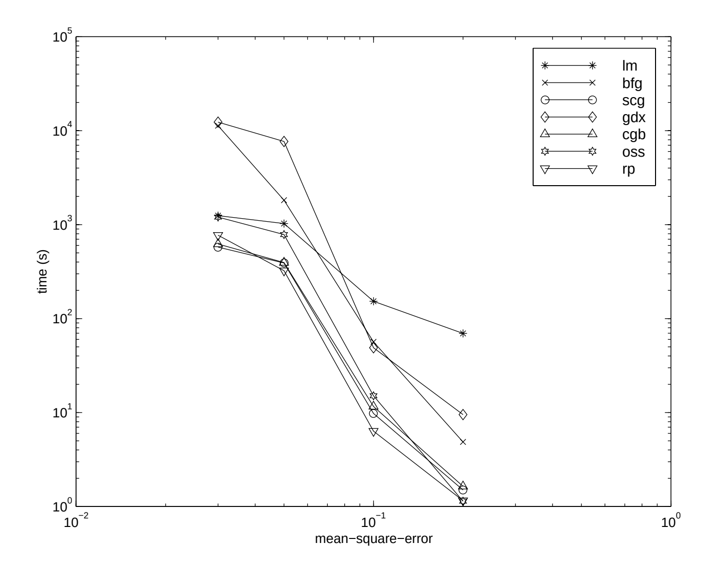
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	323.90	1.00	187.43	576.90	111.37
SCG	390.53	1.21	267.99	487.17	75.07
CGB	394.67	1.22	312.25	558.21	85.38
CGP	415.90	1.28	320.62	614.62	94.77
OSS	784.00	2.42	706.89	936.52	76.37
CGF	784.50	2.42	629.42	1082.20	144.63
LM	1028.10	3.17	802.01	1269.50	166.31

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
BFG	1821.00	5.62	1415.80	3254.50	546.36
GDX	7687.00	23.73	5169.20	10350.00	2015.00

The following figure plots the mean square error versus execution time for some typical algorithms. As with other problems, we see that the SCG and RP have fast initial convergence, while the LM algorithm is able to provide smaller final error.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. In this case, we can see that the BFG algorithm degrades as the error goal is reduced, while the LM algorithm improves. The RP algorithm is best, except at the smallest error goal, where SCG is better.



Summary

There are several algorithm characteristics that we can deduce from the experiments we have described. In general, on function approximation problems, for networks that contain up to a few hundred weights, the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. In many cases, `trainlm` is able to obtain lower mean square errors than any of the other algorithms tested. However, as the number of weights in the network increases, the advantage of the `trainlm` decreases. In addition, `trainlm` performance is relatively poor on pattern recognition problems. The storage requirements of `trainlm` are larger than the other algorithms tested. By adjusting the `mem_reduc` parameter, discussed earlier, the storage requirements can be reduced, but at a cost of increased execution time.

The `trainrp` function is the fastest algorithm on pattern recognition problems. However, it does not perform well on function approximation problems. Its performance also degrades as the error goal is reduced. The memory requirements for this algorithm are relatively small in comparison to the other algorithms considered.

The conjugate gradient algorithms, in particular `trainscg`, seem to perform well over a wide variety of problems, particularly for networks with a large

number of weights. The SCG algorithm is almost as fast as the LM algorithm on function approximation problems (faster for large networks) and is almost as fast as `trainrp` on pattern recognition problems. Its performance does not degrade as quickly as `trainrp` performance does when the error is reduced. The conjugate gradient algorithms have relatively modest memory requirements.

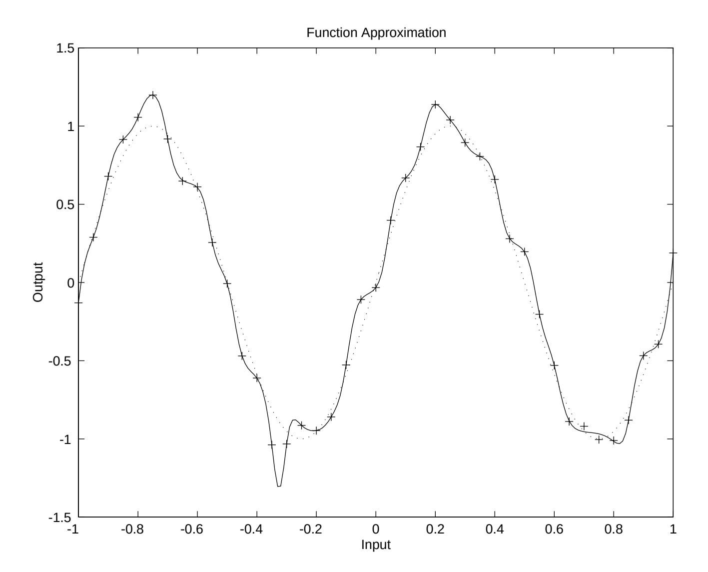
The `trainbfg` performance is similar to that of `trainlm`. It does not require as much storage as `trainlm`, but the computation required does increase geometrically with the size of the network, since the equivalent of a matrix inverse must be computed at each iteration.

The variable learning rate algorithm `traingdx` is usually much slower than the other methods, and has about the same storage requirements as `trainrp`, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping (as described in the next section) you may have inconsistent results if you use an algorithm that converges too quickly. You may overshoot the point at which the error on the validation set is minimized.

Improving Generalization

One of the problems that occurs during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network that has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the '+' symbols, and the neural network response is given by the solid line. Clearly this network has overfit the data and will not generalize well.



One method for improving network generalization is to use a network that is just large enough to provide an adequate fit. The larger a network you use, the more complex the functions the network can create. If we use a small enough network, it will not have enough power to overfit the data. Run the Neural Network Design Demonstration `nnd11gn` [HDB96] to investigate how reducing the size of a network can prevent overfitting.

Unfortunately, it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving

generalization that are implemented in the Neural Network Toolbox: regularization and early stopping. The next few subsections describe these two techniques, and the routines to implement them.

Note that if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. If you can easily collect more data and increase the size of the training set, then there is no need to worry about the following techniques to prevent overfitting. The rest of this section only applies to those situations in which you want to make the most of a limited supply of data.

Regularization

The first method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next subsection explains how the performance function can be modified, and the following subsection describes a routine that automatically sets the optimal performance function to achieve the best generalization.

Modified Performance Function

The typical performance function that is used for training feedforward neural networks is the mean sum of squares of the network errors.

$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

It is possible to improve generalization if we modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases

$$msereg = \gamma mse + (1-\gamma)msw$$

where γ is the performance ratio, and

$$msw = \frac{1}{n}\sum_{j=1}^{n} w_j^2$$

Using this performance function will cause the network to have smaller weights and biases, and this will force the network response to be smoother and less likely to overfit.

In the following code we reinitialize our previous network and retrain it using the BFGS algorithm with the regularized performance function. Here we set the performance ratio to 0.5, which gives equal weight to the mean square errors and the mean square weights.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If we make this parameter too large, we may get overfitting. If the ratio is too small, the network will not adequately fit the training data. In the next section we describe a routine that automatically sets the regularization parameters.

Automated Regularization (trainbr)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. We can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this users guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97].

Bayesian regularization has been implemented in the function `trainbr`. The following code shows how we can train a 1-20-1 network using this function to approximate the noisy sine wave shown earlier in this section.

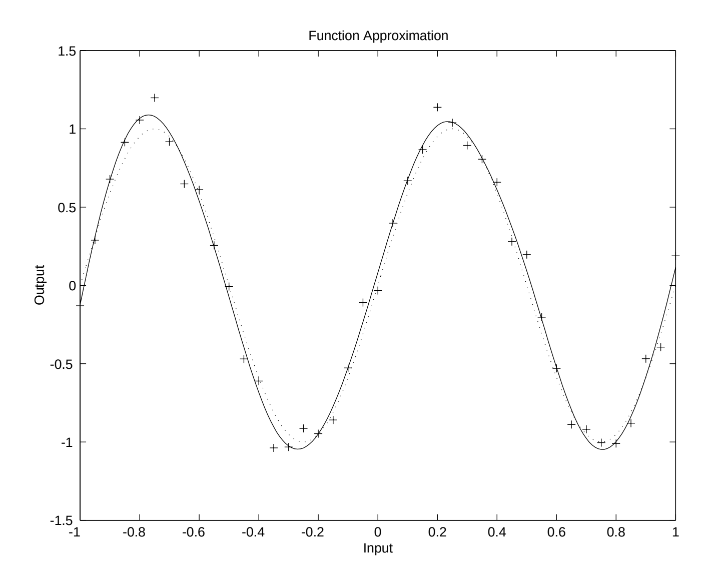
```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net=newff(minmax(p),[20,1],{'tansig','purelin'},'trainbr');
net.trainParam.show = 10;
net.trainParam.epochs = 50;
randn('seed',192736547);
net = init(net);
[net,tr]=train(net,p,t);
TRAINBR, Epoch 0/200, SSE 273.764/0, SSW 21460.5, Grad 2.96e+02/
1.00e-10, #Par 6.10e+01/61
TRAINBR, Epoch 40/200, SSE 0.255652/0, SSW 1164.32, Grad 1.74e-02/
1.00e-10, #Par 2.21e+01/61
TRAINBR, Epoch 80/200, SSE 0.317534/0, SSW 464.566, Grad 5.65e-02/
1.00e-10, #Par 1.78e+01/61
TRAINBR, Epoch 120/200, SSE 0.379938/0, SSW 123.028, Grad
3.64e-01/1.00e-10, #Par 1.17e+01/61
TRAINBR, Epoch 160/200, SSE 0.380578/0, SSW 108.294, Grad
6.43e-02/1.00e-10, #Par 1.19e+01/61
```

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by #Par in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the total number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The trainbr algorithm generally works best when the network inputs and targets are scaled so that they fall approximately in the range [-1,1]. That is the case for the test problem we have used. If your inputs and targets do not fall in this range, you can use the functions premnmx, or prestd, to perform the scaling, as described later in this chapter.

The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfit the data, here we see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. We could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using `trainbr`, it is important to let the algorithm run until the effective number of parameters has converged. The training may stop with the message "Maximum MU reached." This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively constant over several iterations. When this occurs you may want to push the "Stop Training" button in the training window.



Early Stopping

Another method for improving generalization is called *early stopping*. In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error will normally decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set will typically begin to rise. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during the training, but it is used to compare different models. It is also useful to plot the test set error during the training

process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this may indicate a poor division of the data set.

Early stopping can be used with any of the training functions that were described earlier in this chapter. You simply need to pass the validation data to the training function. The following sequence of commands demonstrates how to use the early stopping function.

First we create a simple test problem. For our training set we generate a noisy sine wave with input points ranging from -1 to 1 at steps of 0.05.

```
p = [-1:0.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Next we generate the validation set. The inputs range from -1 to 1, as in the test set, but we offset them slightly. To make the problem more realistic, we also add a different noise sequence to the underlying sine wave. Notice that the validation set is contained in a structure that contains both the inputs and the targets.

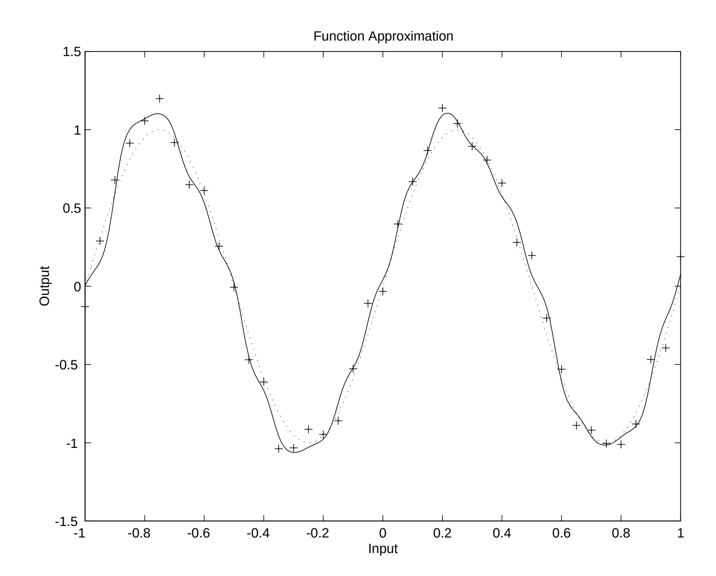
```
val.P = [-0.975:.05:0.975];
val.T = sin(2*pi*v.P)+0.1*randn(size(v.P));
```

We now create a 1-20-1 network, as in our previous example with regularization, and train it. (Notice that the validation structure is passed to `train` after the initial input and layer conditions, which are null vectors in this case since the network contains no delays. Also, in this example we are not using a test set. The test set structure would be the next argument in the call to train.) For this example we use the training function `traingdx`, although early stopping can be used with any of the other training functions we have discussed in this chapter.

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'traingdx');
net.trainParam.show = 25;
net.trainParam.epochs = 300;
net = init(net);
[net,tr]=train(net,p,t,[],[],val);
TRAINGDX, Epoch 0/300, MSE 9.39342/0, Gradient 17.7789/1e-06
TRAINGDX, Epoch 25/300, MSE 0.312465/0, Gradient 0.873551/1e-06
TRAINGDX, Epoch 50/300, MSE 0.102526/0, Gradient 0.206456/1e-06
TRAINGDX, Epoch 75/300, MSE 0.0459503/0, Gradient 0.0954717/1e-06
TRAINGDX, Epoch 100/300, MSE 0.015725/0, Gradient 0.0299898/1e-06
```

```
TRAINGDX, Epoch 125/300, MSE 0.00628898/0, Gradient 0.042467/
1e-06
TRAINGDX, Epoch 131/300, MSE 0.00650734/0, Gradient 0.133314/
1e-06
TRAINGDX, Validation stop.
```

The following figure shows a graph of the network response. We can see that the network did not overfit the data, as in the earlier example, although the response is not extremely smooth, as when using regularization. This is characteristic of early stopping.



Summary and Discussion

Both regularization and early stopping can ensure network generalization when properly applied.

When using Bayesian regularization, it is important to train the network until it reaches convergence. The sum squared error, the sum squared weights, and the effective number of parameters should reach constant values when the network has converged.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), you want to set the

training parameters so that the convergence is relatively slow (e.g., set `mu` to a relatively large value, such as 1, and set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively). The training functions `trainscg` and `trainrp` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

With both regularization and early stopping, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

Based on our experience, Bayesian regularization generally provides better generalization performance than early stopping, when training function approximation networks. This is because Bayesian regularization does not require that a validation data set be separated out of the training data set. It uses all of the data. This advantage is especially noticeable when the size of the data set is small.

To provide you with some insight into the performance of the algorithms, we tested both early stopping and Bayesian regularization on several benchmark data sets, which are listed in the following table.

Data Set Title	**No. pts.**	**Network**	**Description**
BALL	67	2-10-1	Dual-sensor calibration for a ball position measurement.
SINE (5% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 5% level.
SINE (2% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 2% level.
ENGINE (ALL)	119 9	2-30-2	Engine sensor - full data set.
ENGINE (1/4)	300	2-30-2	Engine sensor - 1/4 of data set.

Data Set Title	No. pts.	Network	Description
CHOLEST (ALL)	264	5-15-3	Cholesterol measurement - full data set.
CHOLEST (1/2)	132	5-15-3	Cholesterol measurement - 1/2 data set.

These data sets are of various sizes, with different numbers of inputs and targets. With two of the data sets we trained the networks once using all of the data and then retrained the networks using only a fraction of the data. This illustrates how the advantage of Bayesian regularization becomes more noticeable when the data sets are smaller. All of the data sets are obtained from physical systems, except for the SINE data sets. These two were artificially created by adding various levels of noise to a single cycle of a sine wave. The performance of the algorithms on these two data sets illustrates the effect of noise.

The following table summarizes the performance of Early Stopping (ES) and Bayesian Regularization (BR) on the seven test sets. (The `trainscg` algorithm was used for the early stopping tests. Other algorithms provide similar performance.)

Mean Squared Test Set Error

Method	Ball	Engine (All)	Engine (1/4)	Choles (All)	Choles (1/2)	Sine (5% N)	Sine (2% N)
ES	1.2e-1	1.3e-2	1.9e-2	1.2e-1	1.4e-1	1.7e-1	1.3e-1
BR	1.3e-3	2.6e-3	4.7e-3	1.2e-1	9.3e-2	3.0e-2	6.3e-3
ES/BR	92	5	4	1	1.5	5.7	21

We can see that Bayesian regularization performs better than early stopping in most cases. The performance improvement is most noticeable when the data set is small, or if there is little noise in the data set. The BALL data set, for example, was obtained from sensors that had very little noise.

Although the generalization performance of Bayesian regularization is often better than early stopping, this is not always the case. In addition, the form of

Bayesian regularization implemented in the toolbox does not perform as well on pattern recognition problems as it does on function approximation problems. This is because the approximation to the Hessian that is used in the Levenberg-Marquardt algorithm is not as accurate when the network output is saturated, as would be the case in pattern recognition problems. Another disadvantage of the Bayesian regularization method is that it generally takes longer to converge than early stopping.

Preprocessing and Postprocessing

Neural network training can be made more efficient if certain preprocessing steps are performed on the network inputs and targets. In this section, we describe several preprocessing routines that you can use.

Min and Max (premnmx, postmnmx, tramnmx)

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function premnmx can be used to scale inputs and targets so that they fall in the range [-1,1]. The following code illustrates the use of this function.

```
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net=train(net,pn,tn);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets, pn and tn, that are returned will all fall in the interval [-1,1]. The vectors minp and maxp contain the minimum and maximum values of the original inputs, and the vectors mint and maxt contain the minimum and maximum values of the original targets. After the network has been trained, these vectors should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If premnmx is used to scale both the inputs and targets, then the output of the network will be trained to produce outputs in the range [-1,1]. If you want to convert these outputs back into the same units that were used for the original targets, then you should use the routine postmnmx. In the following code, we simulate the network that was trained in the previous code, and then convert the network output back into the original units.

```
an = sim(net,pn);
a = postmnmx(an,mint,maxt);
```

The network output an will correspond to the normalized targets tn. The un-normalized network output a is in the same units as the original targets t.

If premnmx is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums that were computed for the training set. This can be

accomplished with the routine tramnmx. In the following code, we apply a new set of inputs to the network we have already trained.

```
pnewn = tramnmx(pnew,minp,maxp);
anewn = sim(net,pnewn);
anew = postmnmx(anewn,mint,maxt);
```

Mean and Stand. Dev. (prestd, poststd, trastd)

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. This procedure is implemented in the function prestd. It normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of prestd.

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets, pn and tn, that are returned will have zero means and unity standard deviation. The vectors meanp and stdp contain the mean and standard deviations of the original inputs, and the vectors meant and stdt contain the means and standard deviations of the original targets. After the network has been trained, these vectors should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If prestd is used to scale both the inputs and targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. If you want to convert these outputs back into the same units that were used for the original targets, then you should use the routine poststd. In the following code we simulate the network that was trained in the previous code, and then convert the network output back into the original units.

```
an = sim(net,pn);
a = poststd(an,meant,stdt);
```

The network output an corresponds to the normalized targets tn. The un-normalized network output a is in the same units as the original targets t.

If prestd is used to preprocess the training set data, then whenever the trained network is used with new inputs, they should be preprocessed with the means and standard deviations that were computed for the training set. This can be

accomplished with the routine trastd. In the following code, we apply a new set of inputs to the network we have already trained.

```
pnewn = trastd(pnew,meanp,stdp);
anewn = sim(net,pnewn);
anew = poststd(anewn,meant,stdt);
```

Principal Component Analysis (prepca, trapca)

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the components of the input vectors (so that they are uncorrelated with each other); it orders the resulting orthogonal components (principal components) so that those with the largest variation come first; and it eliminates those components that contribute the least to the variation in the data set. The following code illustrates the use of prepca, which performs a principal component analysis.

```
[pn,meanp,stdp] = prestd(p);
[ptrans,transMat] = prepca(pn,0.02);
```

Note that we first normalize the input vectors, using prestd, so that they have zero mean and unity variance. This is a standard procedure when using principal components. In this example, the second argument passed to prepca is 0.02. This means that prepca eliminates those principal components that contribute less than 2% to the total variation in the data set. The matrix ptrans contains the transformed input vectors. The matrix transMat contains the principal component transformation matrix. After the network has been trained, this matrix should be used to transform any future inputs that are applied to the network. It effectively becomes a part of the network, just like the network weights and biases. If you multiply the normalized input vectors pn by the transformation matrix transMat, you obtain the transformed input vectors ptrans.

If prepca is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the transformation matrix that was computed for the training set. This can be accomplished with the routine trapca. In the following code, we apply a new set of inputs to a network we have already trained.

```
pnewn = trastd(pnew,meanp,stdp);
pnewtrans = trapca(pnewn,transMat);
a = sim(net,pnewtrans);
```

Post-Training Analysis (postreg)

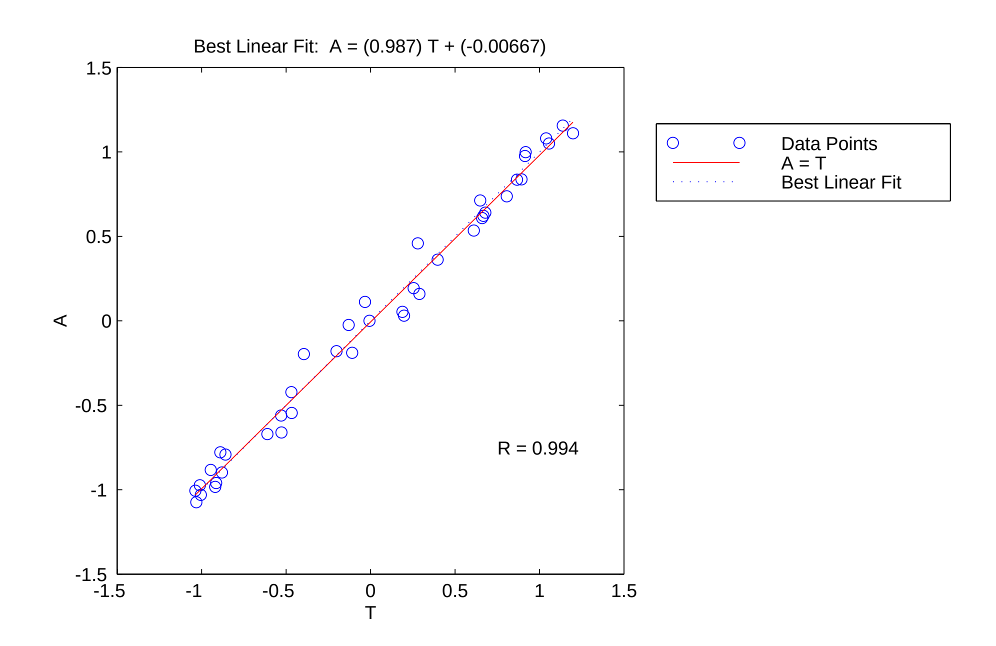
The performance of a trained network can be measured to some extent by the errors on the training, validation and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine `postreg` is designed to perform this analysis.

The following commands illustrate how we can perform a regression analysis on the network that we previously trained in the early stopping section.

```
a = sim(net,p);
[m,b,r] = postreg(a,t)
m =
    0.9874
b =
   -0.0067
r =
    0.9935
```

Here we pass the network output and the corresponding targets to `postreg`. It returns three parameters. The first two, `m` and `b`, correspond to the slope and the y-intercept of the best linear regression relating targets to network outputs. If we had a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. In this example, we can see that the numbers are very close. The third variable returned by postreg is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In our example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by `postreg`. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from the perfect fit line, because the fit is so good.



Best Linear Fit: A = (0.987) T + (-0.00667)
1.5
1
0.5
0
-0.5
-1
-1.5
A
-1.5
-1
-0.5
0
0.5
1
1.5
T
R = 0.994
Data Points
A = T
Best Linear Fit

Sample Training Session

We have covered a number of different concepts in this chapter. At this point it might be useful to put some of these ideas together with an example of how a typical training session might go.

For this example, we are going to use data from a medical application [PuLu92]. We want to design an instrument that can determine serum cholesterol levels from measurements of spectral content of a blood sample. We have a total of 264 patients for which we have measurements of 21 wavelengths of the spectrum. For the same patients we also have measurements of hdl, ldl, and vldl cholesterol levels, based on serum separation. The first step is to load the data into the workspace and perform a principal component analysis.

```
load choles_all
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.001);
```

Here we have conservatively retained those principal components which account for 99.9% of the variation in the data set. Let's check the size of the transformed data.

```
[R,Q] = size(ptrans)
R =
  4
Q =
  264
```

There was apparently significant redundancy in the data set, since the principal component analysis has reduced the size of the input vectors from 21 to 4.

The next step is to divide the data up into training, validation and test subsets. We will take one fourth of the data for the validation set, one fourth for the test set and one half for the training set. We pick the sets as equally spaced points throughout the original data.

```
iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q 3:4:Q];
val.P = ptrans(:,iival); val.T = tn(:,iival);
test.P = ptrans(:,iitst); test.T = tn(:,iitst);
ptr = ptrans(:,iitr); ttr = tn(:,iitr);
```

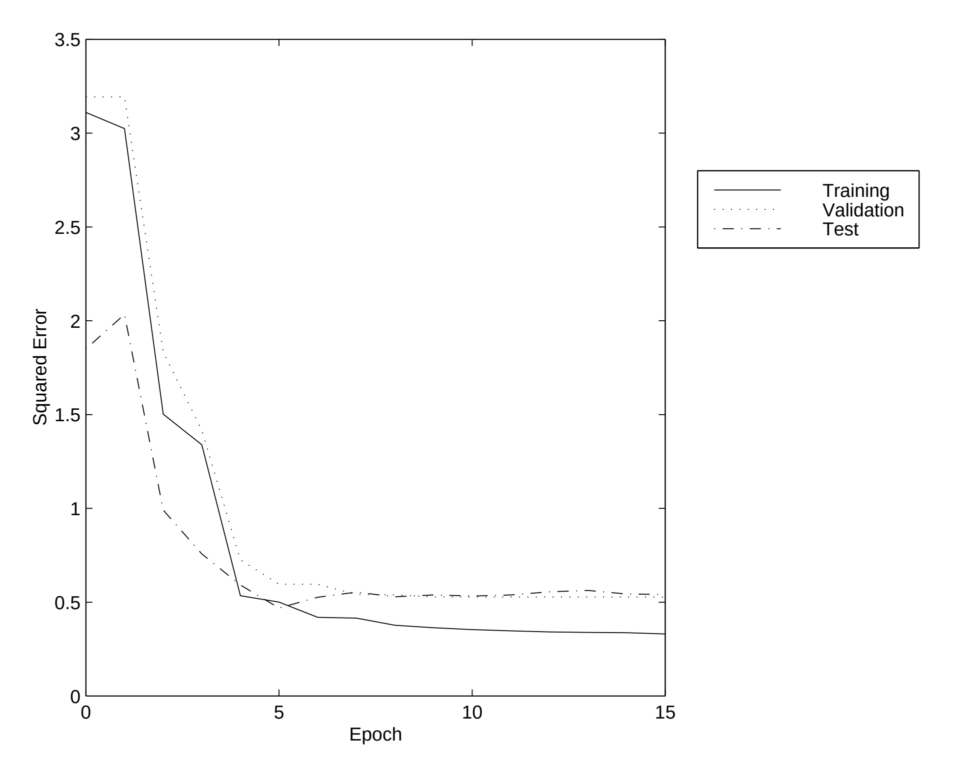
We are now ready to create a network and train it. For this example, we will try a two-layer network, with tan-sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This is a useful structure for function approximation (or regression) problems. As an initial guess, we use five neurons in the hidden layer. The network should have three output neurons since there are three targets. We will use the Levenberg-Marquardt algorithm for training.

```
net = newff(minmax(ptr),[5 3],{'tansig' 'purelin'},'trainlm');
[net,tr]=train(net,ptr,ttr,[],[],val,test);
TRAINLM, Epoch 0/100, MSE 3.11023/0, Gradient 804.959/1e-10
TRAINLM, Epoch 15/100, MSE 0.330295/0, Gradient 104.219/1e-10
TRAINLM, Validation stop.
```

The training stopped after 15 iterations because the validation error increased. It is a useful diagnostic tool to plot the training, validation and test errors to check the progress of training. We can do that with the following commands.

```
plot(tr.epoch,tr.perf,tr.epoch,tr.vperf,tr.epoch,tr.tperf)
legend('Training','Validation','Test',-1);
ylabel('Squared Error'); xlabel('Epoch')
```

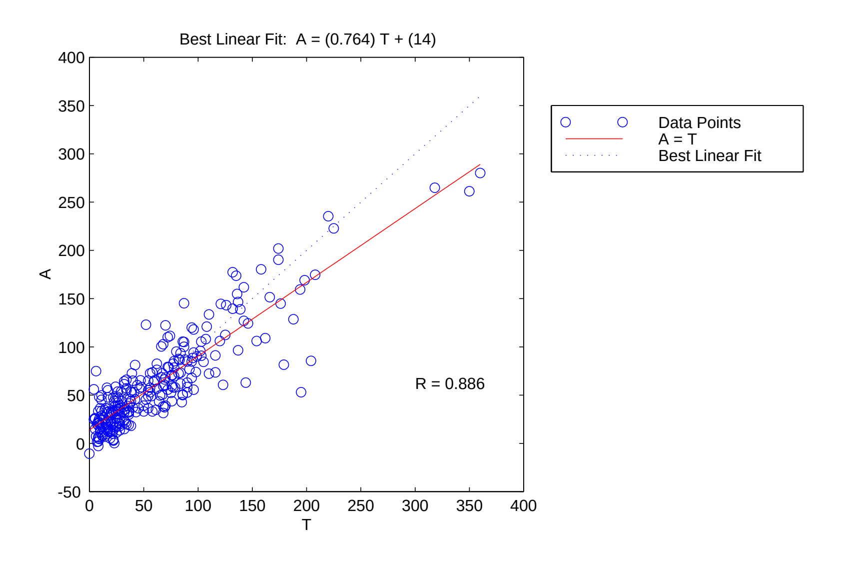
The result is shown in the following figure. The result here is reasonable, since the test set error and the validation set error have similar characteristics, and it doesn’t appear that any significant overfitting has occurred.



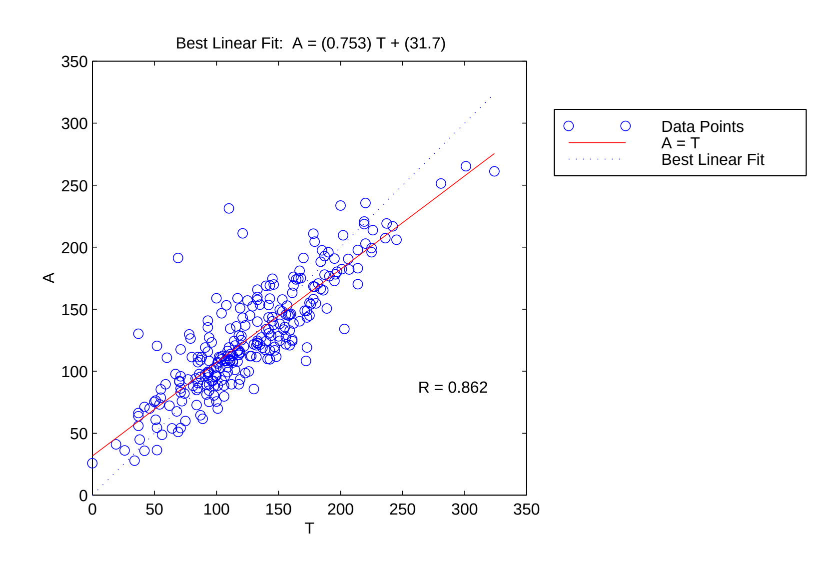
The next step is to perform some analysis of the network response. We will put the entire data set through the network (training, validation and test) and will perform a linear regression between the network outputs and the corresponding targets. First we need to unnormalize the network outputs.

```
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
for i=1:3
  figure(i)
  [m(i),b(i),r(i)] = postreg(a(i,:),t(i,:));
end
```

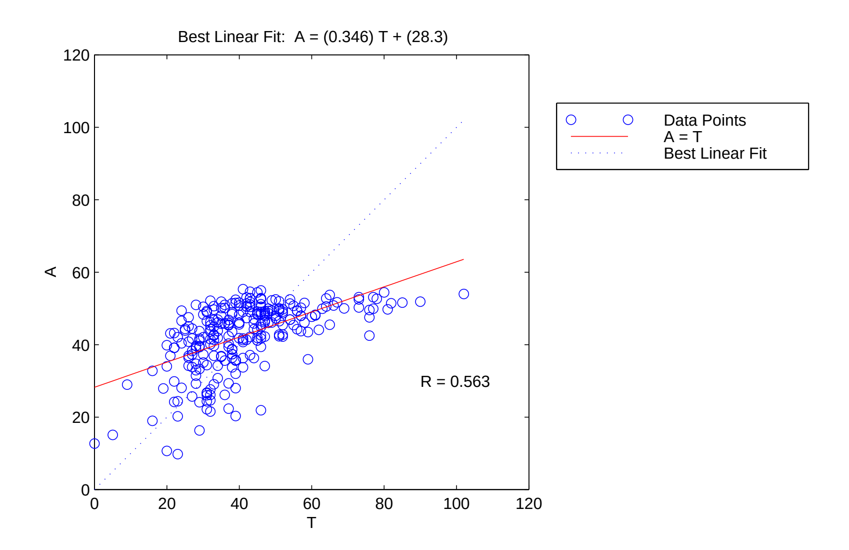
In this case, we have three outputs, so we perform three regressions. The results are shown in the following figures.



Best Linear Fit: A = (0.764) T + (14)
A
T
R = 0.886
Data Points
A = T
Best Linear Fit
400
350
300
250
200
150
100
50
0
-50
0
50
100
150
200
250
300
350
400



Best Linear Fit: A = (0.753) T + (31.7)
A
T
R = 0.862
Data Points
A = T
Best Linear Fit
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300
350



The first two outputs seem to track the targets reasonably well (this is a difficult problem), and the R-values are almost 0.9. The third output (vldl levels) is not well modeled. We probably need to work more on that problem. We might go on to try other network architectures (more hidden layer neurons), or to try Bayesian regularization instead of early stopping for our training technique. Of course there is also the possibility that vldl levels cannot be accurately computed based on the given spectral components.

The function `demobp1` contains a Slide show demonstration of the sample training session. The function `nnsample1` contains all of the commands that we used in this section. You can use it as a template for your own training sessions.

Limitations and Cautions

The gradient descent algorithm is generally very slow because it requires small learning rates for stable learning. The momentum variation is usually faster than simple gradient descent, since it allows higher learning rates while maintaining stability, but it is still too slow for many practical applications. These two methods would normally be used only when incremental training is desired. You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrp`.

Multi-layered networks are capable of performing just about any linear or nonlinear computation, and can approximate any reasonable function arbitrarily well. Such networks overcome the problems associated with the perceptron and linear networks. However, while the network being trained may be theoretically capable of performing correctly, backpropagation and its variations may not always find a solution. See page 12-8 of [HDB96] for a discussion of convergence to local minimum points.

Picking the learning rate for a nonlinear network is a challenge. As with linear networks, a learning rate that is too large leads to unstable learning. Conversely, a learning rate that is too small results in incredibly long training times. Unlike linear networks, there is no easy way of picking a good learning rate for nonlinear multilayer networks. See page 12-8 of [HDB96] for examples of choosing the learning rate. With the faster training algorithms, the default parameter values normally perform adequately.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity see the figures on pages 12-5 to 12-7 of [HDB96], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface it is possible for the network solution to become trapped in one of these local minima. This may happen depending on the initial starting conditions. Settling in a local minimum may be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation will not always find the

correct weights for the optimum solution. You may want to reinitialize the network and retrain several times to guarantee that you have the best solution.

Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fit, but the fitting curve takes wild oscillations between these points. Ways of dealing with various of these issues are discussed in the section on improving generalization. This topic is also discussed starting on page 11-21 of [HDB96].

Summary

Backpropagation can train multilayer feed-forward networks with differentiable transfer functions to perform function approximation, pattern association, and pattern classification. (Other types of networks can be trained as well, although the multilayer network is most commonly used.) The term backpropagation refers to the process by which derivatives of network error, with respect to network weights and biases, can be computed. This process can be used with a number of different optimization strategies.

The architecture of a multilayer network is not completely constrained by the problem to be solved. The number of inputs to the network is constrained by the problem, and the number of neurons in the output layer is constrained by the number of outputs required by the problem. However, the number of layers between network inputs and the output layer and the sizes of the layers are up to the designer.

The two-layer sigmoid/linear network can represent any functional relationship between inputs and outputs if the sigmoid layer has enough neurons.

There are several different backpropagation training algorithms. They have a variety of different computation and storage requirements, and no one algorithm is best suited to all locations. The following list summarizes the training algorithms included in the toolbox.

Function	Description
`traingd`	Basic gradient descent. Slow response, can be used in incremental mode training.
`traingdm`	Gradient descent with momentum. Generally faster than `traingd`. Can be used in incremental mode training.
`traingdx`	Adaptive learning rate. Faster training than `traingd`, but can only be used in batch mode training.
`trainrp`	Resilient backpropagation. Simple batch mode training algorithm with fast convergence and minimal storage requirements.

Function	Description
`traincgf`	Fletcher-Reeves conjugate gradient algorithm. Has smallest storage requirements of the conjugate gradient algorithms.
`traincgp`	Polak-Ribiére conjugate gradient algorithm. Slightly larger storage requirements than `traincgf`. Faster convergence on some problems.
`traincgb`	Powell-Beale conjugate gradient algorithm. Slightly larger storage requirements than `traincgp`. Generally faster convergence.
`trainscg`	Scaled conjugate gradient algorithm. The only conjugate gradient algorithm that requires no line search. A very good general purpose training algorithm.
`trainbfg`	BFGS quasi-Newton method. Requires storage of approximate Hessian matrix and has more computation in each iteration than conjugate gradient algorithms, but usually converges in fewer iterations.
`trainoss`	One step secant method. Compromise between conjugate gradient methods and quasi-Newton methods.
`trainlm`	Levenberg-Marquardt algorithm. Fastest training algorithm for networks of moderate size. Has memory reduction feature for use when the training set is large.
`trainbr`	Bayesian regularization. Modification of the Levenberg-Marquardt training algorithm to produce networks that generalize well. Reduces the difficulty of determining the optimum network architecture.

One problem that can occur when training neural networks is that the network can overfit on the training set and not generalize well to new data outside the training set. This can be prevented by training with `trainbr`, but it can also be prevented by using *early stopping* with any of the other training routines. This requires that the user pass a validation set to the training algorithm, in addition to the standard training set.

To produce the most efficient training, it is often helpful to preprocess the data before training. It is also helpful to analyze the network response after training is complete. The toolbox contains a number of routines for pre- and post-processing. They are summarized in the following table.

Function	Description
`premnmx`	Normalize data to fall in the range [-1,1].
`postmnmx`	Inverse of `premnmx`. Used to convert data back to standard units.
`tramnmx`	Normalize data using previously computed minimums and maximums. Used to preprocess new inputs to networks that have been trained with data normalized with `premnmx`.
`prestd`	Normalize data to have zero mean and unity standard deviation.
`poststd`	Inverse of `prestd`. Used to convert data back to standard units.
`trastd`	Normalize data using previously computed means and standard deviations. Used to preprocess new inputs to networks that have been trained with data normalized with `prestd`.
`prepca`	Principal component analysis. Reduces dimension of input vector and un-correlates components of input vectors.
`trapca`	Preprocess data using previously computed principal component transformation matrix. Used to preprocess new inputs to networks that have been trained with data transformed with `prepca`.
`postreg`	Linear regression between network outputs and targets. Used to determine the adequacy of network fit.

6

Control Systems

Introduction 6-2

NN Predictive Control 6-4
System Identification 6-4
Predictive Control 6-5
Using the NN Predictive Controller Block 6-6

NARMA-L2 (Feedback Linearization) Control 6-14
Identification of the NARMA-L2 Model 6-14
NARMA-L2 Controller 6-16
Using the NARMA-L2 Controller Block 6-18

Model Reference Control 6-23
Using the Model Reference Controller Block 6-25

Importing and Exporting 6-31
Importing and Exporting Networks 6-31
Importing and Exporting Training Data 6-35

Summary 6-38

Introduction

Neural networks have been applied very successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99]. This chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them.

There are typically two steps involved when using neural networks for control:

1. System Identification
2. Control Design

In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this chapter, the system identification stage is identical. The control design stage, however, is different for each architecture.

- For the model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance.
- For the NARMA-L2 control, the controller is simply a rearrangement of the plant model.
- For the model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training.

The next three sections of this chapter discuss model predictive control, NARMA-L2 control and model reference control. Each section consists of a brief

description of the control concept, followed by a demonstration of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink blocks, which are contained in the Neural Network Toolbox blockset.

To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application.

- **Model Predictive Control**. This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form, using any of the training algorithms discussed in Chapter 5. (This is true for all three control architectures.) The controller, however, requires a significant amount of on-line computation, since an optimization algorithm is performed at each sample time to compute the optimal control input.
- **NARMA-L2 Control**. This controller requires the least computation of the three architectures described in this chapter. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (The companion form model is described later in this chapter.)
- **Model Reference Control**. The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained off-line, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [HaJe99]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

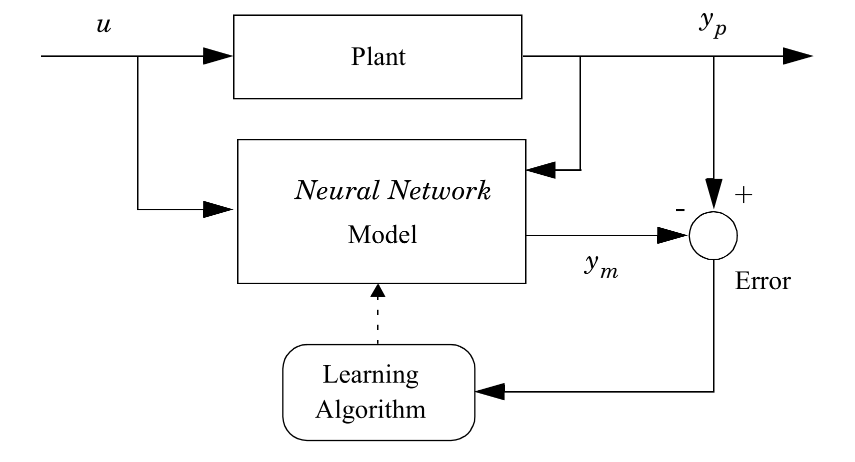
NN Predictive Control

The neural network predictive controller that is implemented in the Neural Network Toolbox uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance. (See the *Model Predictive Control Toolbox Users Guide* for a complete coverage of the application of various model predictive control strategies to linear systems.)

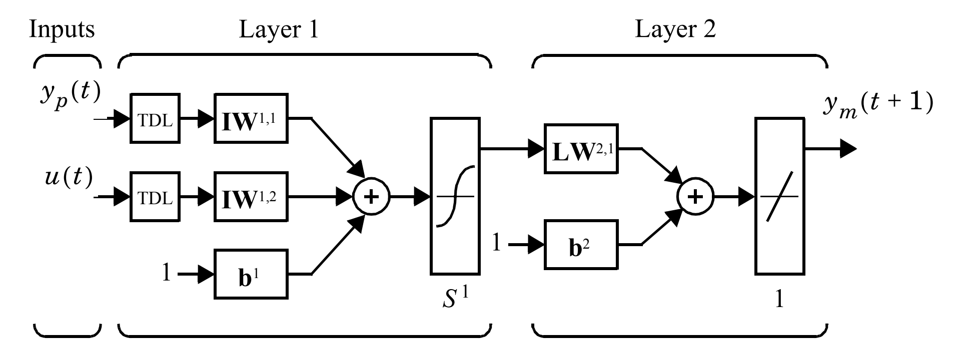
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that has been implemented in Simulink.

System Identification

The first stage of model predictive control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by the following figure.



The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.



This network can be trained offline in batch mode, using data collected from the operation of the plant. Any of the training algorithms discussed in Chapter 5 can be used for network training. This process is discussed in more detail later in this chapter.

Predictive Control

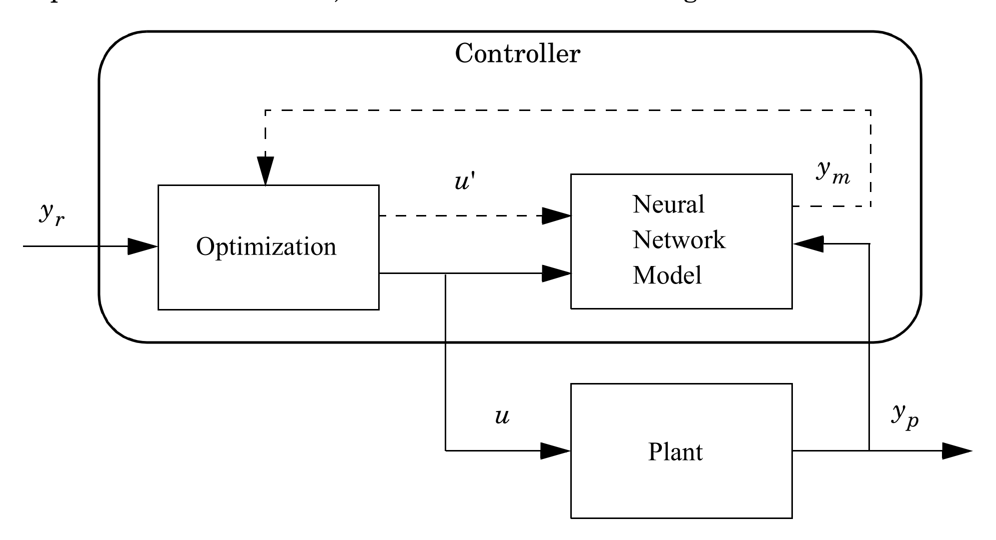
The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon.

$$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$

where N_1, N_2 and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index.

The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and

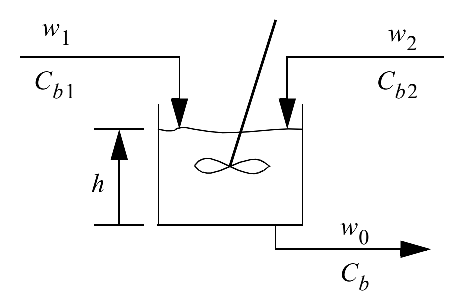
then the optimal u is input to the plant. The controller block has been implemented in Simulink, as described in the following section.



Using the NN Predictive Controller Block

This section demonstrates how the NN Predictive Controller block is used.The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the predictive controller. This demo uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.



The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

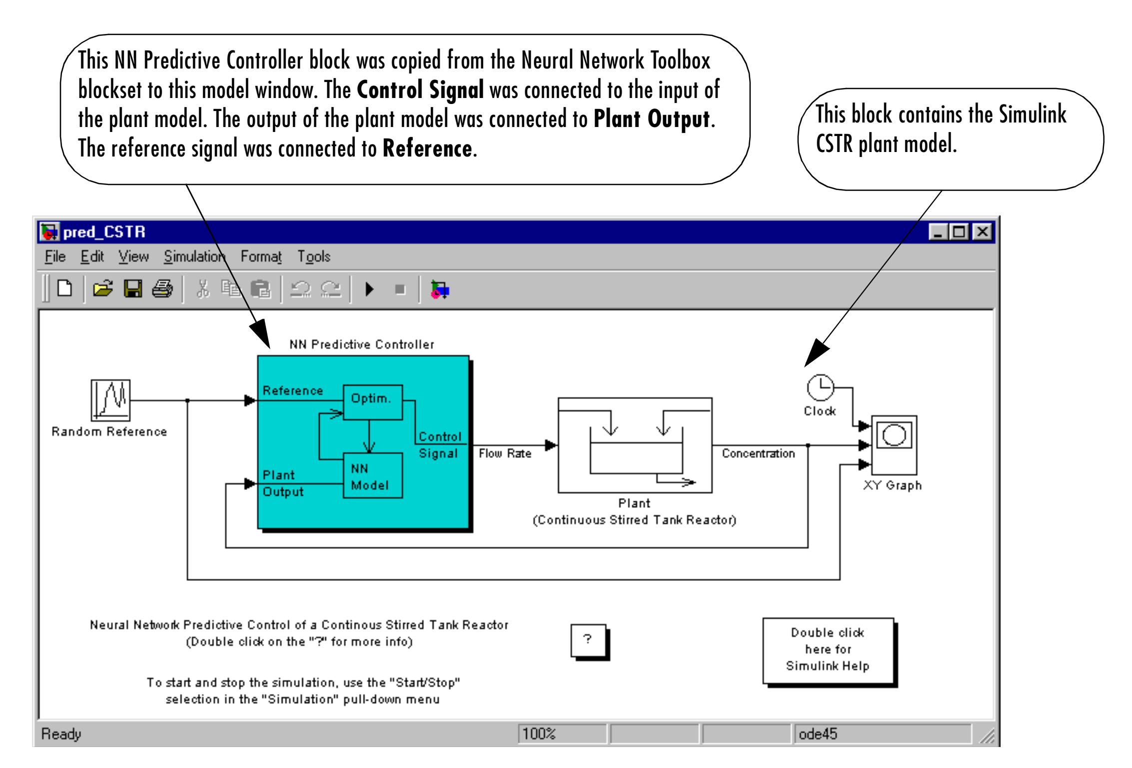
where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1}, and $w_2(t)$ is the flow rate of the diluted feed C_{b2}. The input concentrations are set to $C_{b1} = 24.9$ and $C_{b2} = 0.1$. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$.

The objective of the controller is to maintain the product concentration by adjusting the flow $w_2(t)$. To simplify the demonstration, we set $w_1(t) = 0.1$. The level of the tank $h(t)$ is not controlled for this experiment.

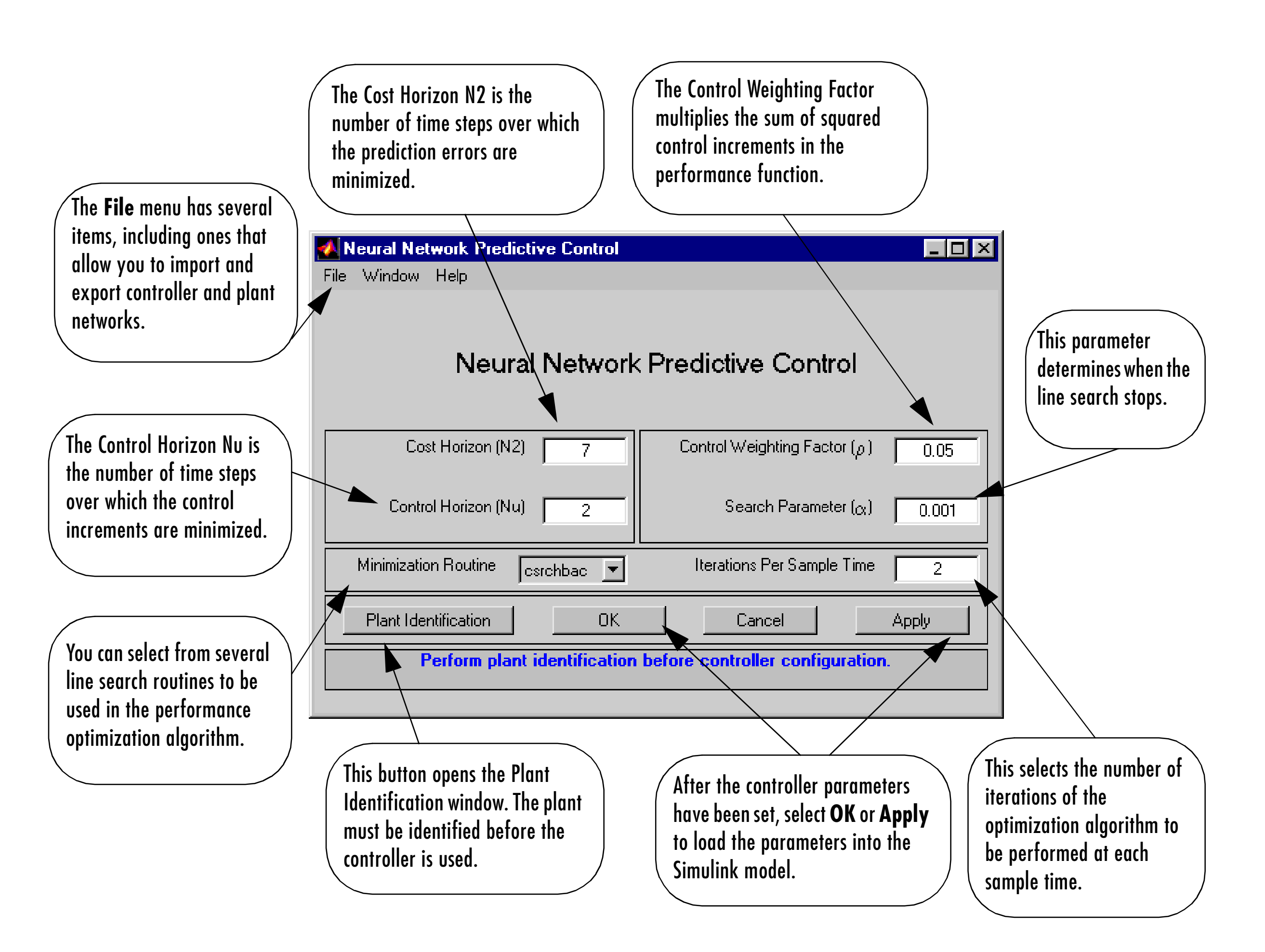
To run this demo, follow these steps.

1 Start MATLAB.

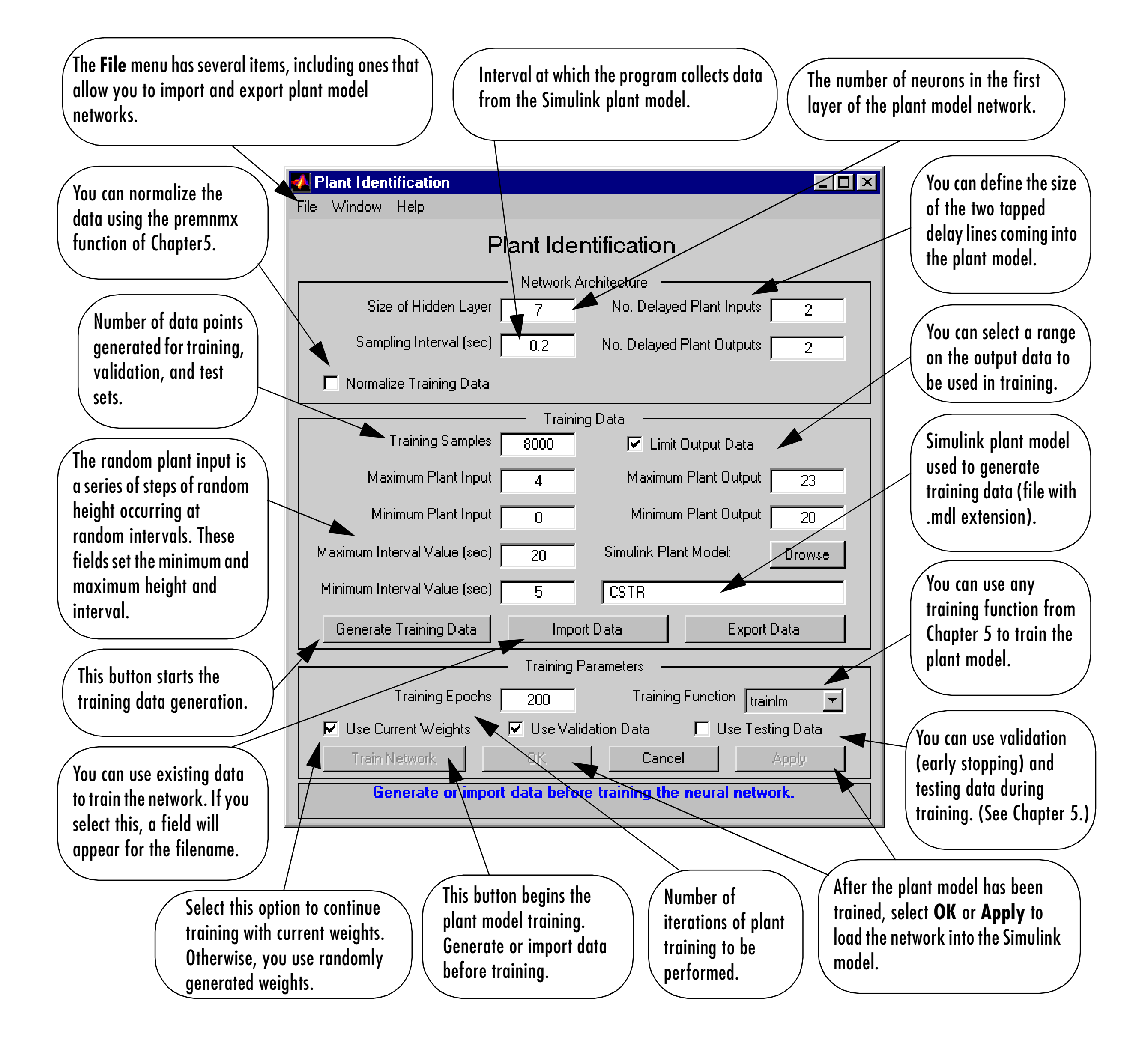
2 Run the demo model by typing `predcstr` in the MATLAB® command window. This command starts Simulink and creates the following model window. The NN Predictive Controller block has already been placed in the model.



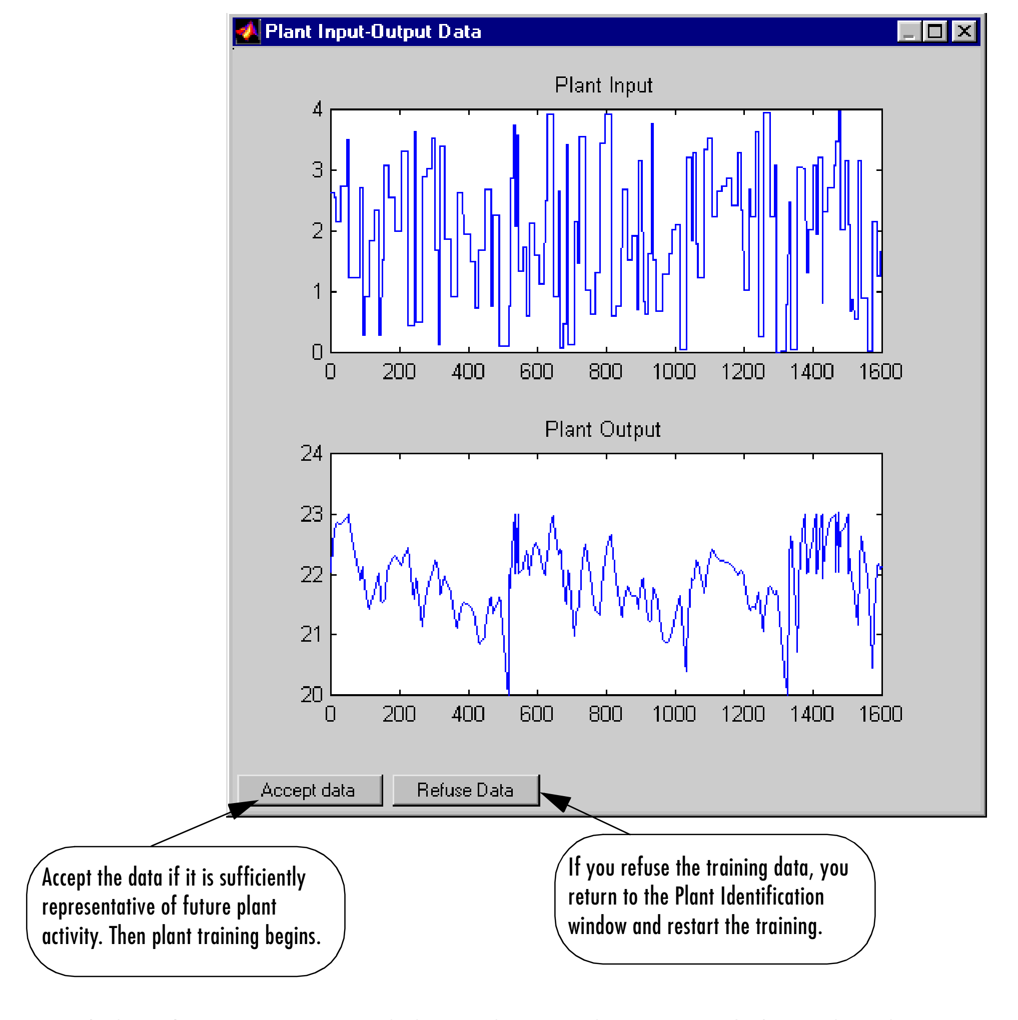
3 Double-click on the NN Predictive Controller block. This brings up the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u. (N_1 is fixed at 1.) The weighting parameter ρ, described earlier, is also defined in this window. The parameter α is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in Chapter 5.



4 Select **Plant Identification**. This opens the following window. The neural network plant model must be developed before the controller is used. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. The size of that layer, the number of delayed inputs and delayed outputs, and the training function are selected in this window. You can select any of the training functions described in Chapter 5 to train the neural network plant model.

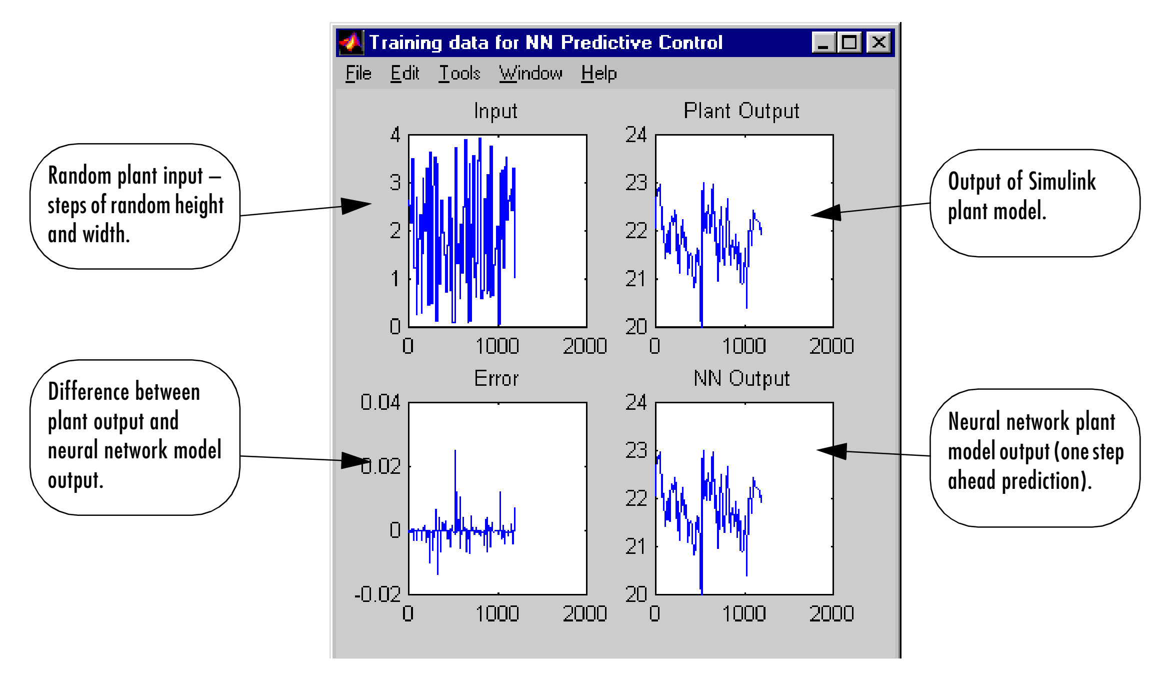

The File menu has several items, including ones that allow you to import and export plant model networks.
Interval at which the program collects data from the Simulink plant model.
The number of neurons in the first layer of the plant model network.
You can normalize the data using the premnmx function of Chapter5.
You can define the size of the two tapped delay lines coming into the plant model.
Number of data points generated for training, validation, and test sets.
You can select a range on the output data to be used in training.
The random plant input is a series of steps of random height occurring at random intervals. These fields set the minimum and maximum height and interval.
Simulink plant model used to generate training data (file with .mdl extension).
This button starts the training data generation.
You can use any training function from Chapter 5 to train the plant model.
You can use existing data to train the network. If you select this, a field will appear for the filename.
You can use validation (early stopping) and testing data during training. (See Chapter 5.)
Select this option to continue training with current weights. Otherwise, you use randomly generated weights.
This button begins the plant model training. Generate or import data before training.
Number of iterations of plant training to be performed.
After the plant model has been trained, select OK or Apply to load the network into the Simulink model.
Plant Identification
File
Window
Help
Plant Identification
Network Architecture
Size of Hidden Layer
7
No. Delayed Plant Inputs
2
Sampling Interval (sec)
0.2
No. Delayed Plant Outputs
2
Normalize Training Data
Training Data
Training Samples
8000
Limit Output Data
Maximum Plant Input
4
Maximum Plant Output
23
Minimum Plant Input
0
Minimum Plant Output
20
Maximum Interval Value (sec)
20
Simulink Plant Model:
Browse
Minimum Interval Value (sec)
5
CSTR
Generate Training Data
Import Data
Export Data
Training Parameters
Training Epochs
200
Training Function
trainlm
Use Current Weights
Use Validation Data
Use Testing Data
Train Network
OK
Cancel
Apply
Generate or import data before training the neural network.

5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.



6 Select **Accept Data**, and then select **Train Network** from the Plant Identification window. Plant model training begins. The training proceeds

according to the selected training algorithm (`trainlm` in this case). This is a straightforward application of batch training, as described in Chapter 5. After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.) You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this demonstration, begin the simulation, as shown in the following steps.

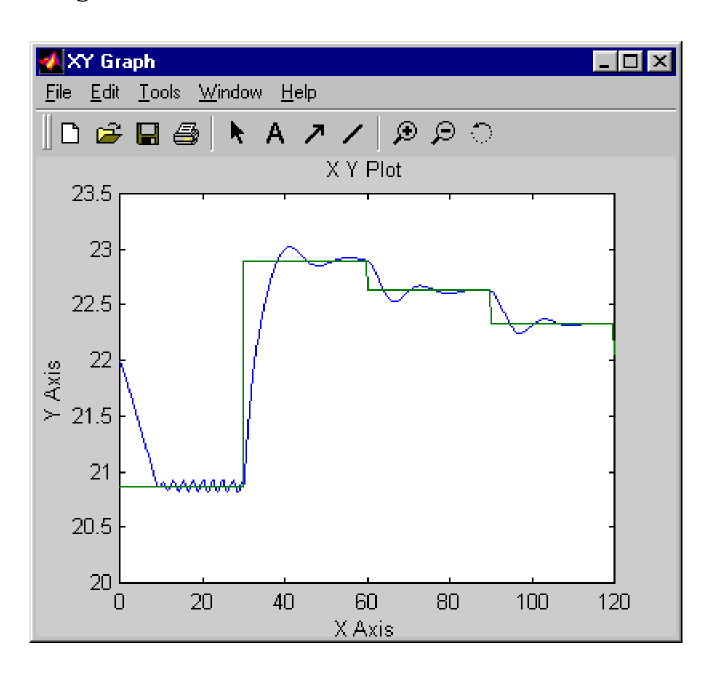


7 Select **OK** in the Plant Identification window. This loads the trained neural network plant model into the NN Predictive Controller block.

8 Select **OK** in the Neural Network Predictive Control window. This loads the controller parameters into the NN Predictive Controller block.

9 Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the

plant output and the reference signal are displayed, as in the following figure.



NARMA-L2 (Feedback Linearization) Control

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form). It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by canceling the nonlinearities. This section begins by presenting the companion form system model and demonstrating how you can use a neural network to identify this model. Then it describes how the identified neural network model can be used to develop a controller. This is followed by a demonstration of how to use the NARMA-L2 Control block, which is contained in the Neural Network Toolbox blockset.

Identification of the NARMA-L2 Model

As with the model predictive control, the first step in using feedback linearization (or NARMA-L2 control) is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that has been used to represent general discrete-time nonlinear systems is the Nonlinear Autoregressive-Moving Average (NARMA) model:

$$y(k+d) = N[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)]$$

where $u(k)$ is the system input, and $y(k)$ is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function N. This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory, $y(k+d) = y_r(k+d)$, the next step is to develop a nonlinear controller of the form:

$$u(k) = G[y(k), y(k-1), \ldots, y(k-n+1), y_r(k+d), u(k-1), \ldots, u(k-m+1)]$$

The problem with using this controller is that if you want to train a neural network to create the function G that will minimize mean square error, you need to use dynamic backpropagation ([NaPa91] or [HaJe99]). This can be quite slow. One solution proposed by Narendra and Mukhopadhyay [NaMu97]

is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\hat{y}(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)$$

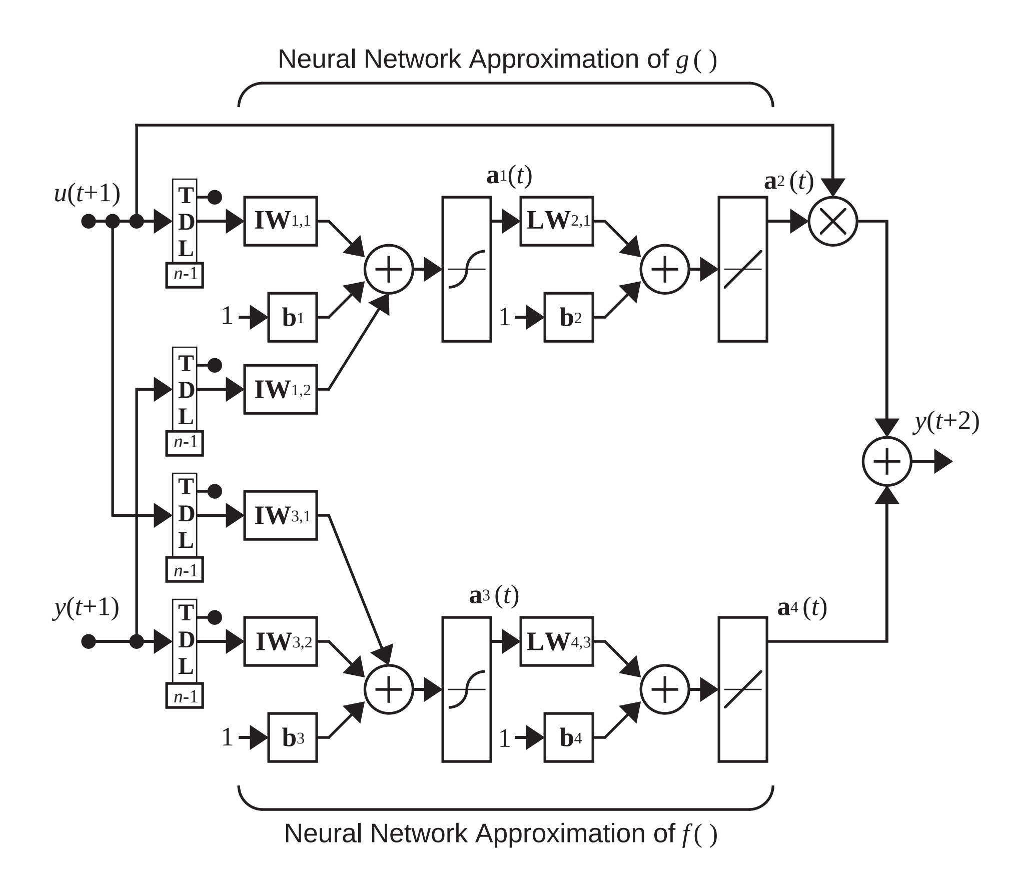
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input $u(k)$ based on the output at the same time, $y(k)$. So, instead, use the model

$$y(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)] + g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)] \cdot u(k+1)$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

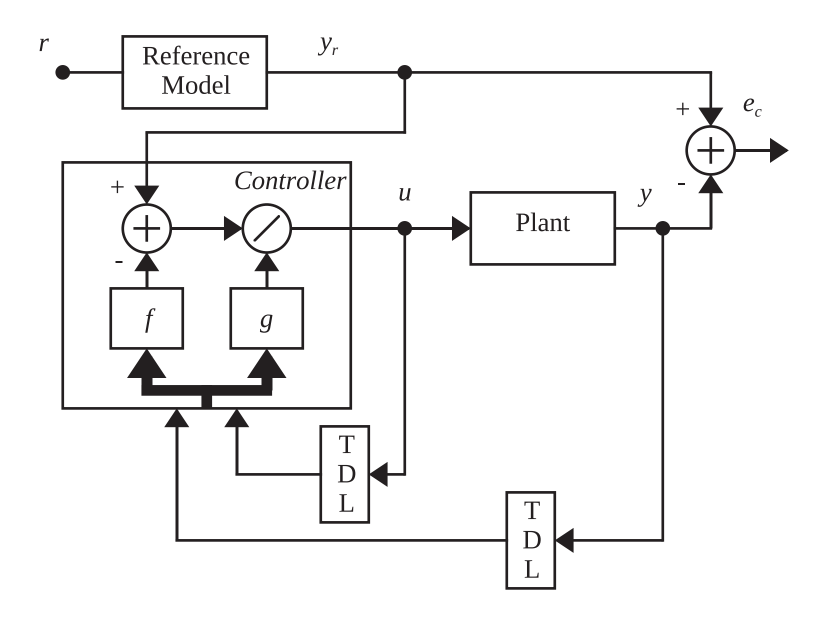


NARMA-L2 Controller

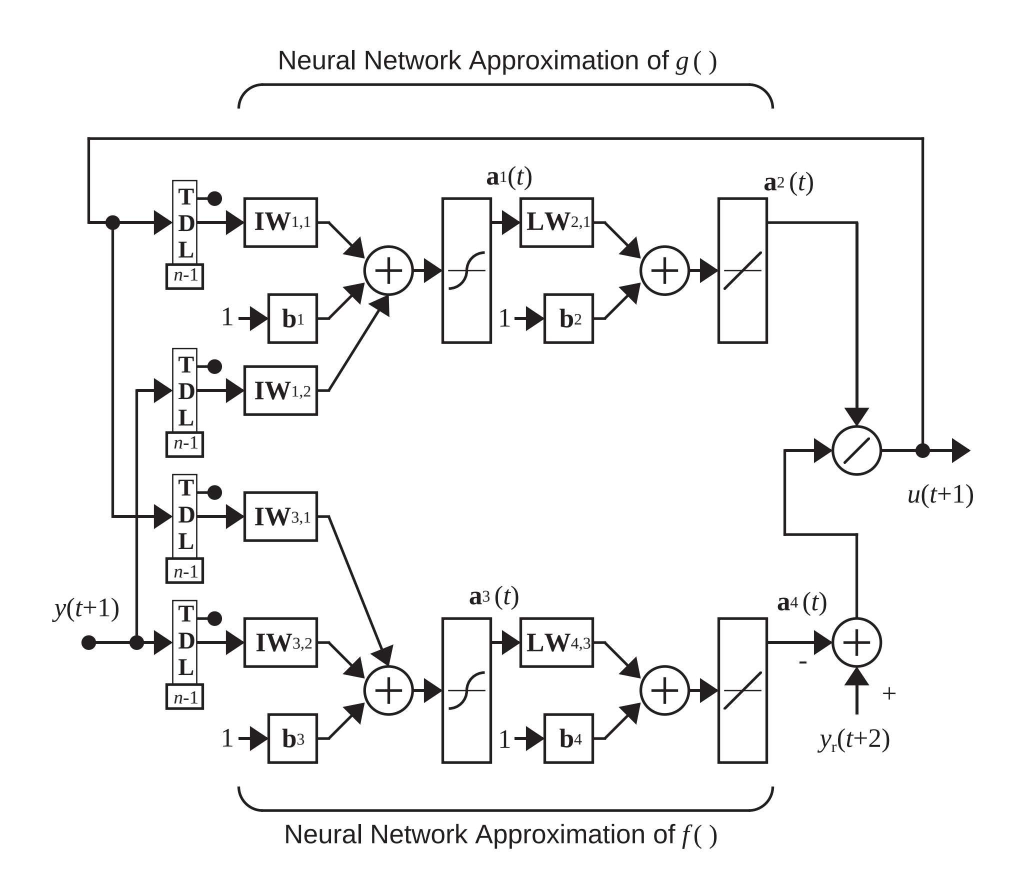
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



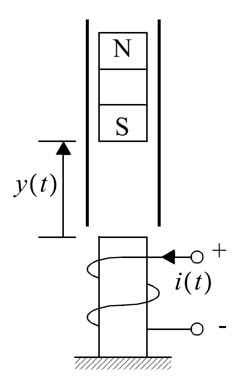
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



Using the NARMA-L2 Controller Block

This section demonstrates how the NARMA-L2 controller is trained.The first step is to copy the NARMA-L2 Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the NARMA-L2 controller. In this demo, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as in the following figure.



The equation of motion for this system is

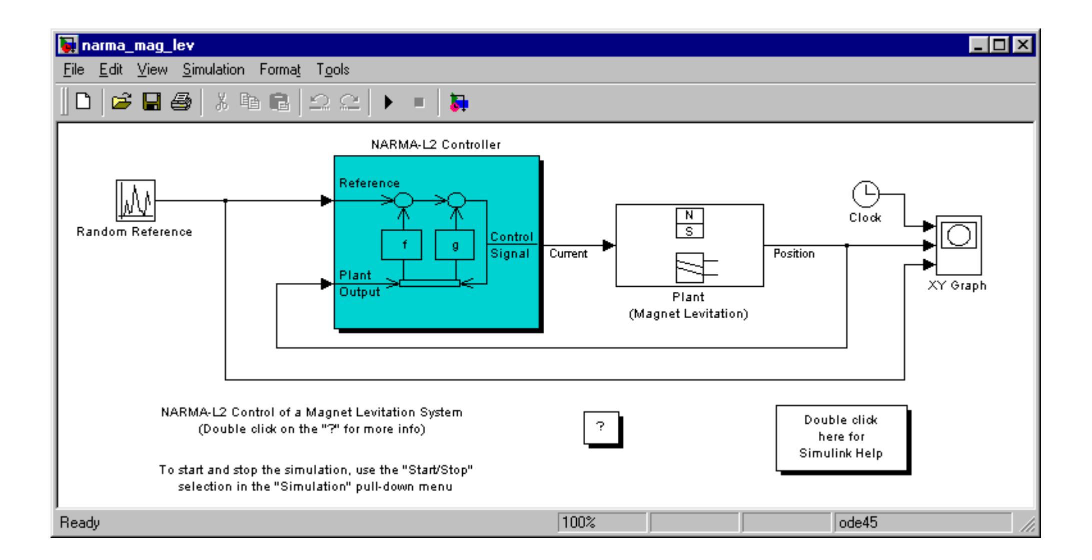
$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.

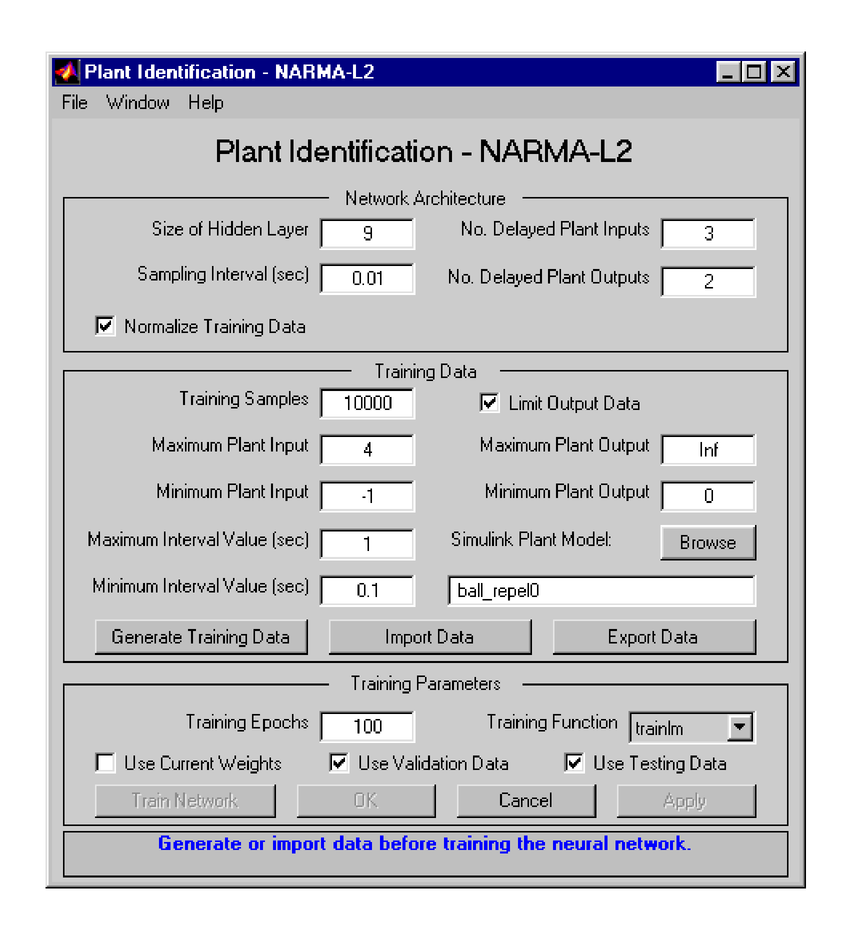
To run this demo, follow these steps.

1. Start MATLAB.

2. Run the demo model by typing `narmamaglev` in the MATLAB command window. This command starts Simulink and creates the following model window. The NARMA-L2 Control block has already been placed in the model.

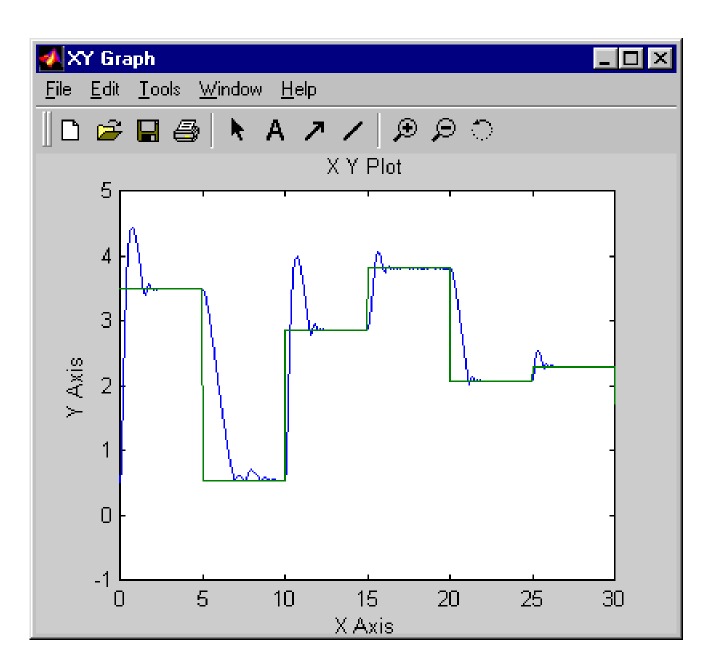


3 Double-click on the NARMA-L2 Controller block. This brings up the following window. Notice that this window enables you to train the NARMA-L2 model. There is no separate window for the controller, since the controller is determined directly from the model, unlike the model predictive controller.



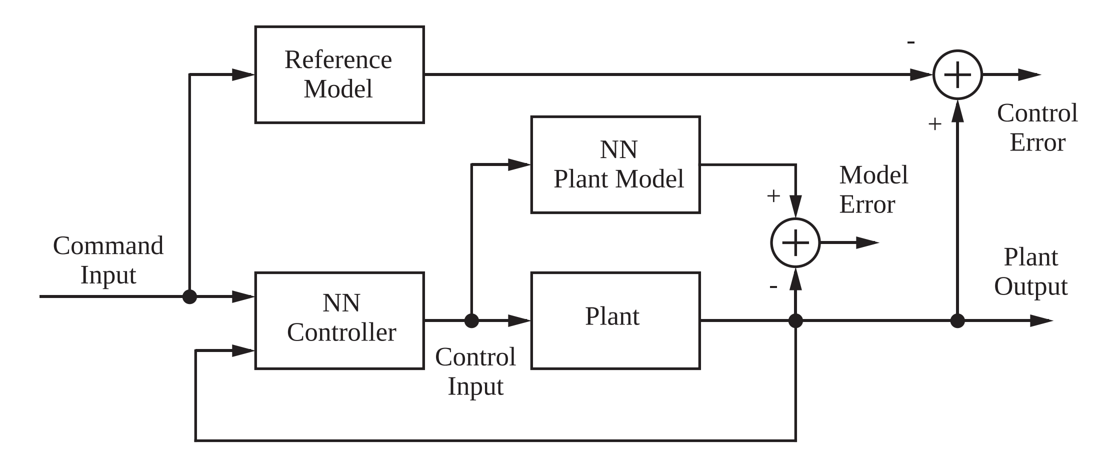
4 Since this window works the same as the other Plant Identification windows, we won't go through the training process again now. Instead, let's simulate the NARMA-L2 controller.

5 Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
5
4
3
2
1
0
-1
Y Axis
0 5 10 15 20 25 30
X Axis

Model Reference Control

The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.



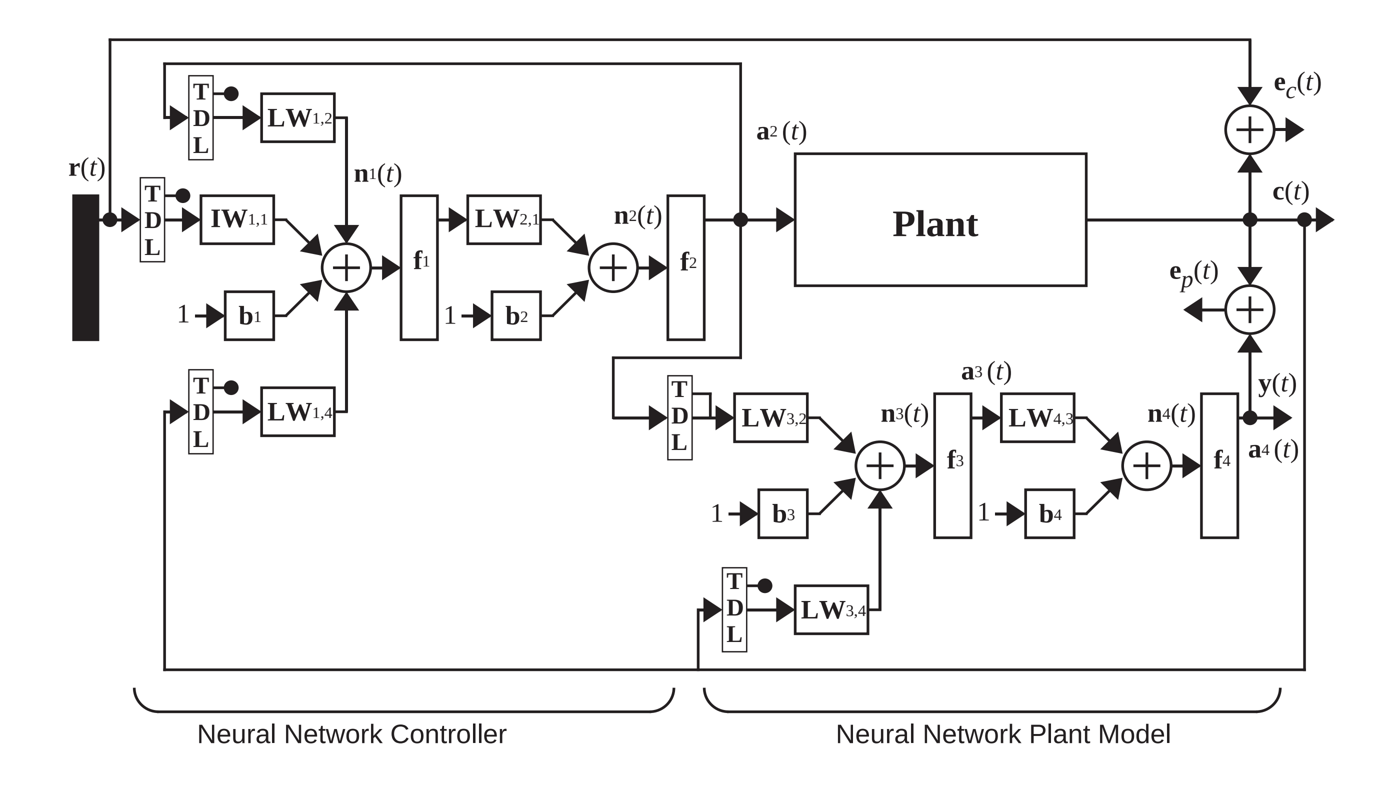
The figure on the following page shows the details of the neural network plant model and the neural network controller, as they are implemented in the Neural Network Toolbox. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs:

- Delayed reference inputs
- Delayed controller outputs
- Delayed plant outputs

For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model:

- Delayed controller outputs
- Delayed plant outputs

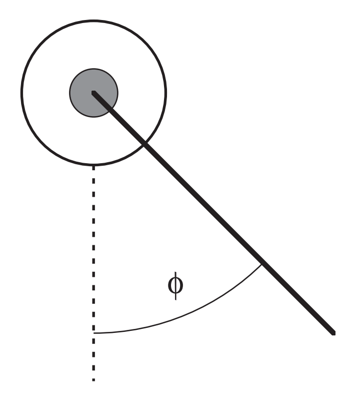
As with the controller, you can set the number of delays. The next section demonstrates how you can set the parameters.


r(t)
T
D
L
IW1,1
LW1,2
LW1,4
1
b1
n1(t)
f1
LW2,1
1
b2
n2(t)
f2
a2 (t)
Plant
ec(t)
c(t)
ep(t)
LW3,2
1
b3
n3(t)
f3
a3 (t)
LW4,3
1
b4
n4(t)
f4
y(t)
a4 (t)
LW3,4
Neural Network Controller
Neural Network Plant Model

Using the Model Reference Controller Block

This section demonstrates how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the model reference controller. In this demo, the objective is to control the movement of a simple, single-link robot arm, as shown in the following figure.



The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10\sin\phi - 2\frac{d\phi}{dt} + u$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor.

The objective is to train the controller so that the arm tracks the reference model

$$\frac{d^2 y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

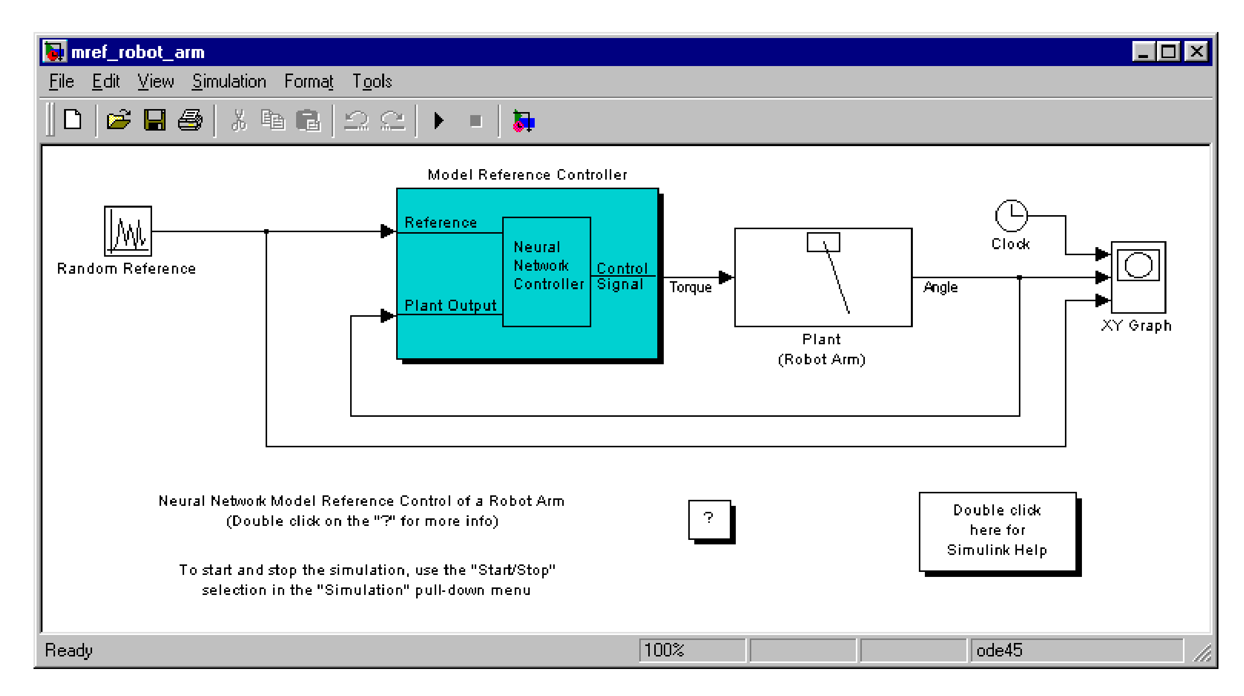
where y_r is the output of the reference model, and r is the input reference signal.

This demo uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

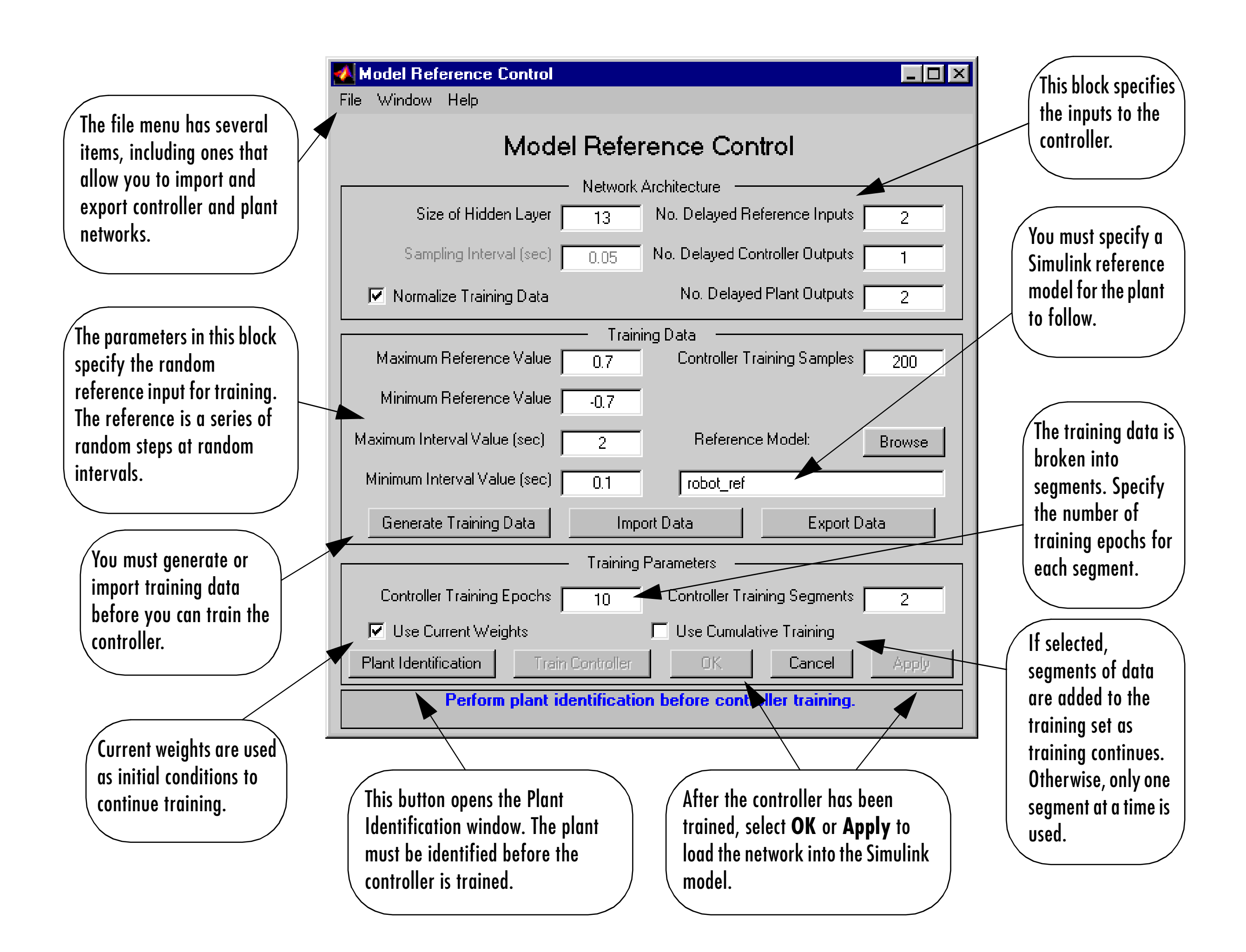
To run this demo, follow these steps.

1 Start MATLAB.

2 Run the demo model by typing `mrefrobotarm` in the MATLAB command window. This command starts Simulink and creates the following model window. The Model Reference Control block has already been placed in the model.

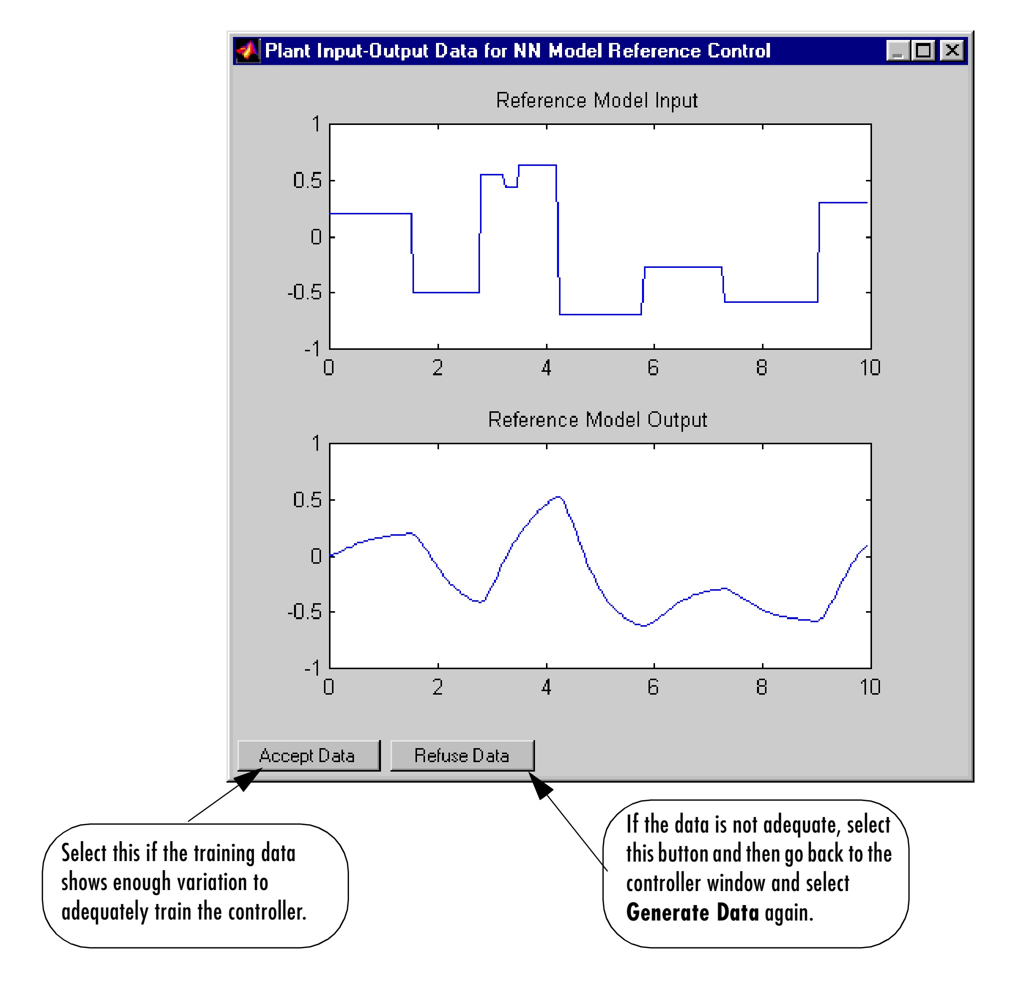


3 Double-click on the Model Reference Control block. This brings up the following window for training the model reference controller.



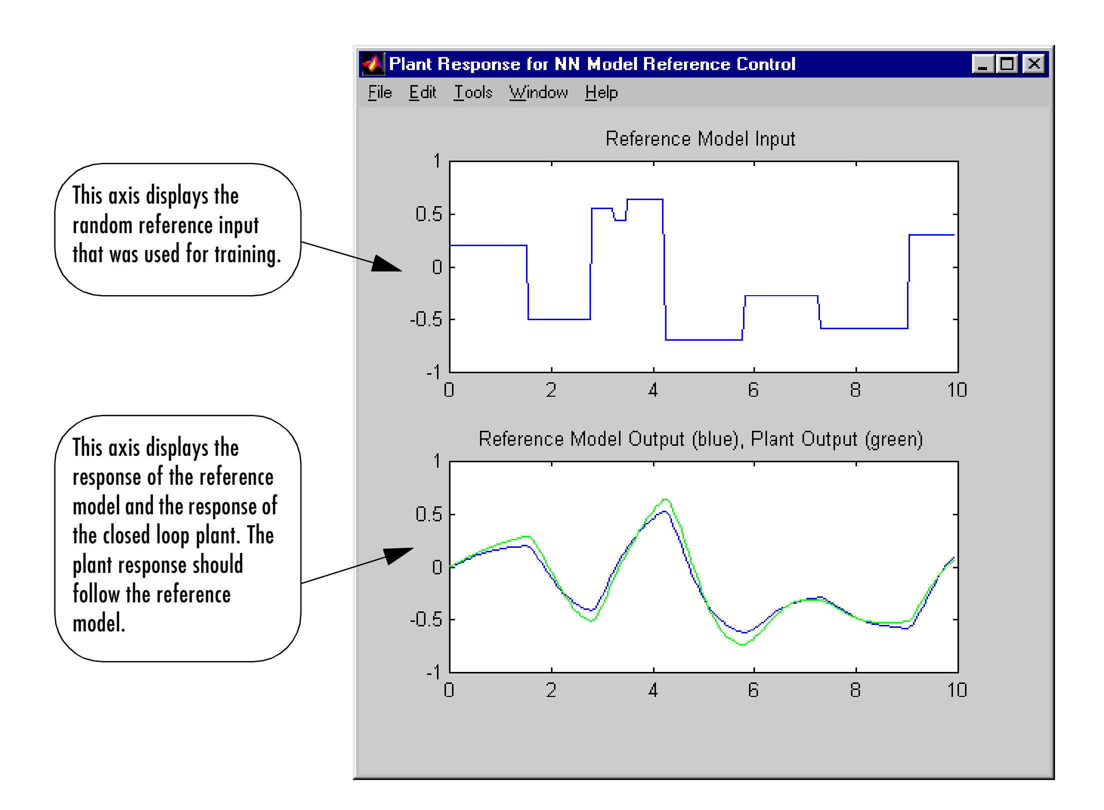
4 The next step would normally be to select **Plant Identification**, which opens the Plant Identification window. You would then train the plant model. Since the Plant Identification window is identical to the one used with the previous controllers, we won't go through that process here.

5 Select **Generate Data**. The program then starts generating the data for training the controller. After the data is generated, the following window appears.



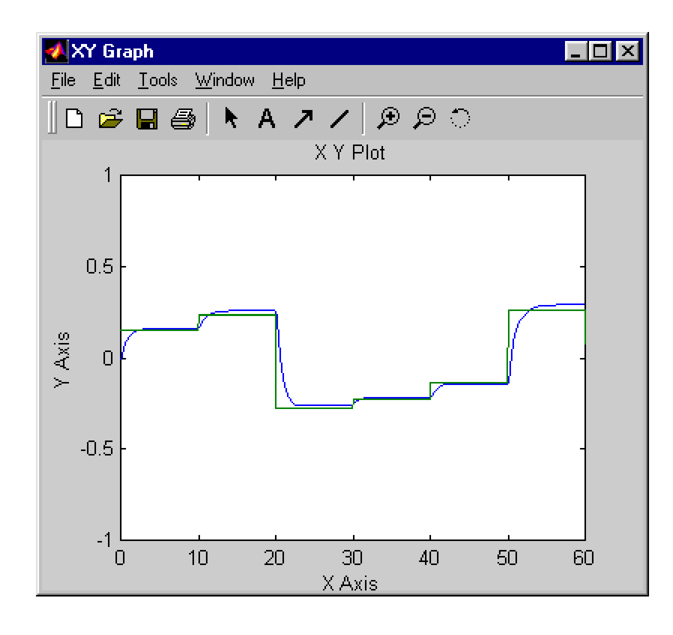
6 Select **Accept Data**. Return to the Model Reference Control window and select **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues one segment at a time until the entire training set has been presented to the network. Controller training can be significantly more time consuming than plant model training. This is

because the controller must be trained using *dynamic* backpropagation (see [HaJe99]). After the training is complete, the response of the resulting closed loop system is displayed, as in the following figure.



7 Go back to the Model Reference Control window. If the performance of the controller is not accurate, then you can select **Train Controller** again, which continues the controller training with the same data set. If you would like to use a new data set to continue training, the select **Generate Data** or **Import Data** before you select **Train Controller**. (Be sure that **Use Current Weights** is selected, if you want to continue training with the same weights.) It may also be necessary to retrain the plant model. If the plant model is not accurate, it can affect the controller training. For this demonstration, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.

8 Return to the Simulink model and start the simulation by selecting the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



Importing and Exporting

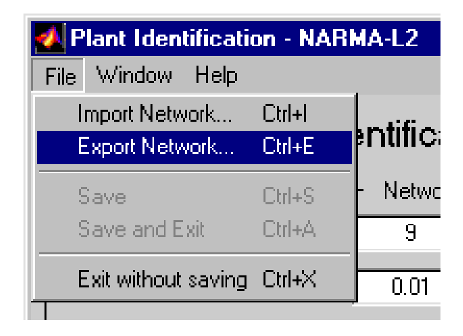
You can save networks and training data to the workspace or to a disk file. The following two sections demonstrate how you can do this.

Importing and Exporting Networks

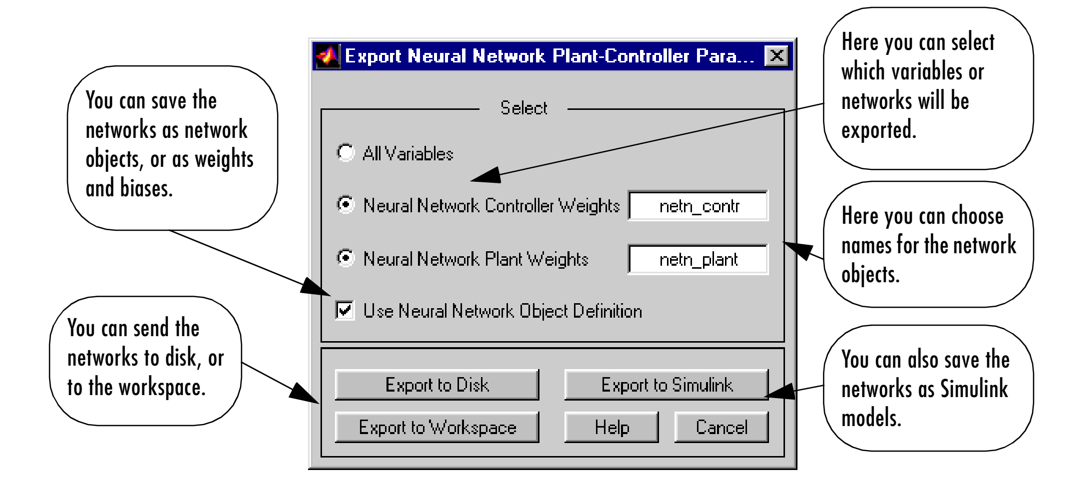
The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you may want to transfer the networks into other applications, or you may want to transfer a network from one controller block to another. You can do this by using the Import Network and Export Network menu commands. The following demonstration leads you through the export and import processes. (We use the NARMA-L2 window for this demonstration, but the same procedure applies to all of the controllers.)

1 Repeat the first three steps of the NARMA-L2 demonstration. The NARMA-L2 Plant Identification window should then be open.

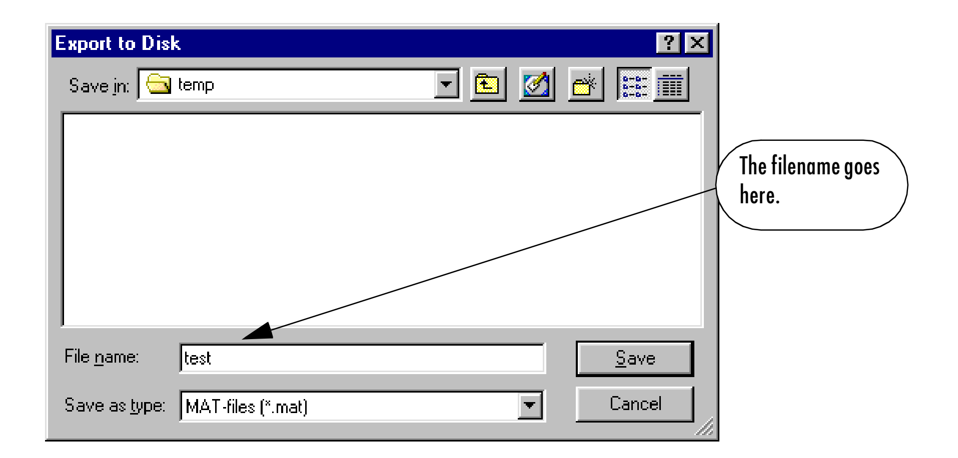
2 Select **Export** from the **File** menu, as shown below.



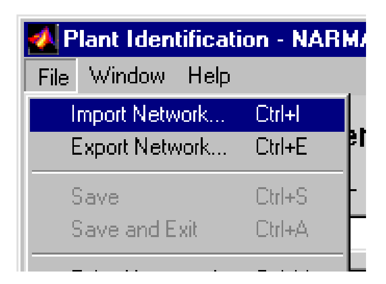
This causes the following window to open.



3 Select **Export to Disk**. The following window opens. Enter the filename `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



4 Retrieve that data with the Import menu command. Select **Import Network** from the **File** menu, as in the following figure.



This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by selecting **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you don't need to import both networks.

Import Neural Network Plant Parameters

Import From: | MAT-file Contents | Neural Network Models

Workspace

MAT-file

MAT-file name:

Browse

Plant

-->

Help | OK | Cancel

Select MAT-file and select **Browse**.

Import file:

Look in: temp

test.mat

File name: test.mat

Files of type: MAT-files (*.mat)

Open | Cancel

Available MAT-files will appear here. Select the appropriate file; then select **Open**.

Import Neural Network Plant Parameters

Import From: | MAT-file Contents | Neural Network Models

Workspace

MAT-file

MAT-file name:

C:\MATLABR11\temp\test

Browse

netn_contr

netn_plant

-->

Plant

netn_plant

Help | OK | Cancel

The available networks appear here.

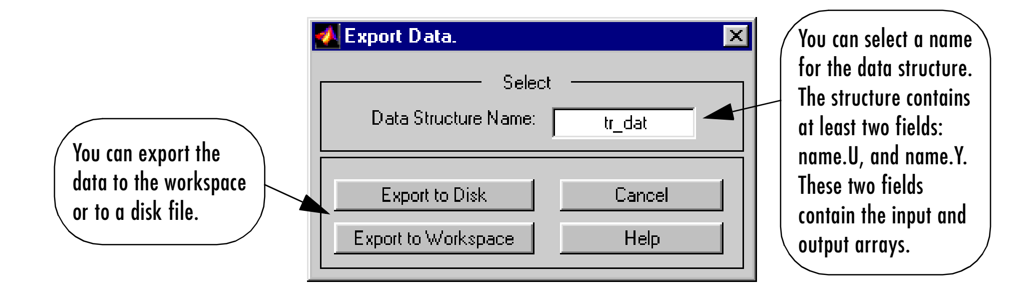
Select the appropriate plant and/or controller and move them into the desired position and select **OK**.

Importing and Exporting Training Data

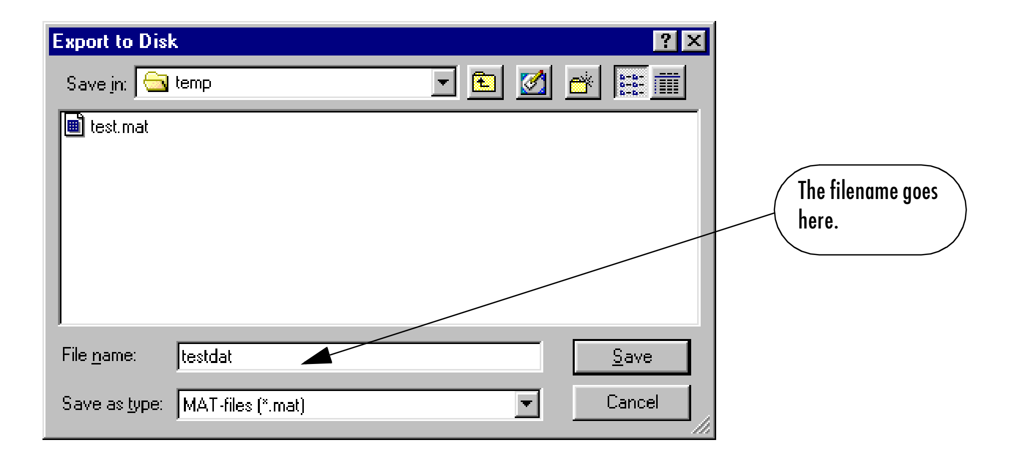
The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You may wish to save the training data to the workspace or to a disk file so that you can load it again at a later time. You may also want to combine data sets manually and then load them back into the training window. You can do this by using the Import and Export buttons. The following demonstration leads you through the import and export processes. (We use the NN Predictive Control window for this demonstration, but the same procedure applies to all of the controllers.)

1. Repeat the first five steps of the NN Predictive Control demonstration. Then select **Accept Data**. The Plant Identification window should then be open, and the Import and Export buttons should be active.

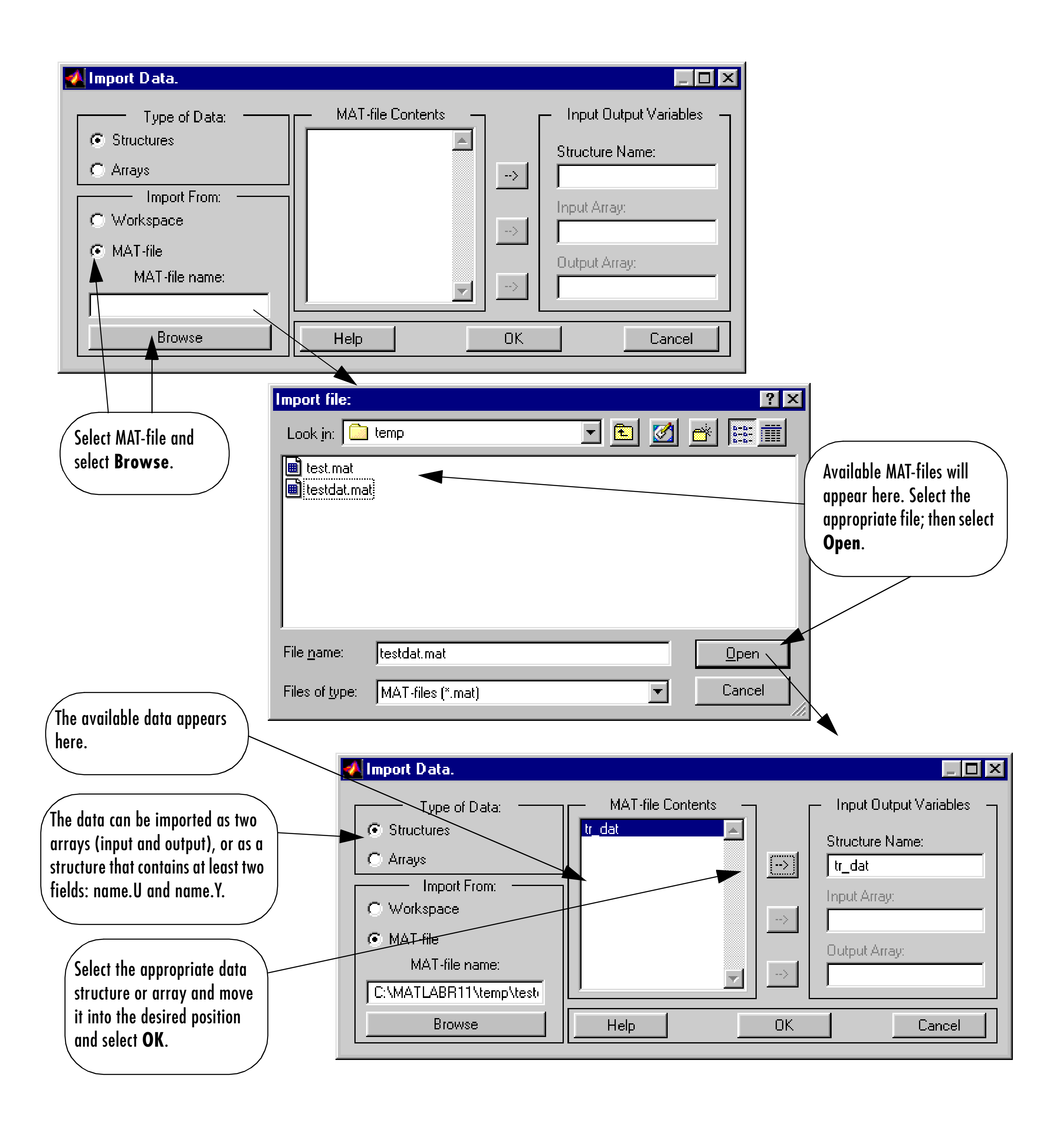
2. Select the **Export** button. This causes the following window to open.



3. Select **Export to Disk**. The following window opens. Enter the filename `testdat` in the box, and select **Save**. This saves the training data structure to disk.



4 Now let's retrieve the data with the import command. Select the **Import** button in the Plant Identification window.This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.


Import Data.
Type of Data:
Structures
Arrays
Import From:
Workspace
MAT-file
MAT-file name:
Browse
MAT-file Contents
Input Output Variables
Structure Name:
Input Array:
Output Array:
Help
OK
Cancel
Select MAT-file and select **Browse**.
Import file:
Look in:
temp
test.mat
testdat.mat
Available MAT-files will appear here. Select the appropriate file; then select **Open**.
File name:
testdat.mat
Open
Files of type:
MAT-files (*.mat)
Cancel
The available data appears here.
The data can be imported as two arrays (input and output), or as a structure that contains at least two fields: name.U and name.Y.
Select the appropriate data structure or array and move it into the desired position and select **OK**.
tr_dat
C:\MATLABR11\temp\testi

Summary

The following table summarizes the controllers discussed in this chapter.

Block	Description
NN Predictive Control	Uses a neural network plant model to predict future plant behavior. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for static networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other controllers.
NARMA-L2 Control	An approximate plant model is in companion form. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation and is reasonably fast. The controller is a rearrangement of the plant model, and requires minimal online computation.
Model Reference Control	A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation.

7

Radial Basis Networks

Introduction 7-2
Important Radial Basis Functions 7-2

Radial Basis Functions 7-3
Neuron Model 7-3
Network Architecture 7-4
Exact Design (newrbe) 7-5
More Efficient Design (newrb) 7-7
Demonstrations 7-8

Generalized Regression Networks 7-9
Network Architecture 7-9
Design (newgrnn) 7-10

Probabilistic Neural Networks 7-12
Network Architecture 7-12
Design (newpnn) 7-13

Summary 7-15
Figures 7-16
New Functions 7-18

Introduction

Radial basis networks may require more neurons than standard feed-forward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feed-forward networks. They work best when many training vectors are available.

You may want to consult the following paper on this subject:

Chen, S., C.F.N. Cowan, and P. M. Grant, "Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, vol. 2, no. 2, March 1991, pp. 302-309.

This chapter discusses two variants of radial basis networks, Generalized Regression networks (GRNN) and Probabilistic neural networks (PNN). You may want to read about them in P.D. Wasserman, *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993 on pp. 155-61, and pp. 35-55 respectively.

Important Radial Basis Functions

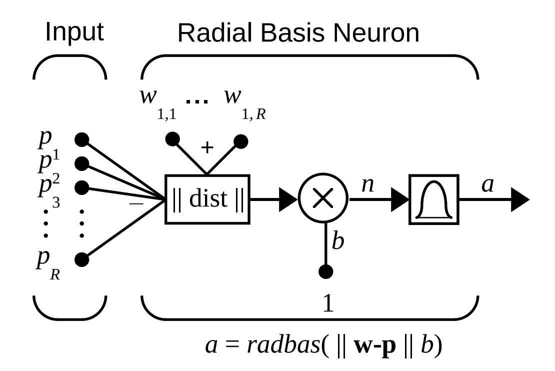
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNN and PNN can be designed with `newgrnn` and `newpnn`, respectively.

Type `help radbasis` to see a listing of all functions and demonstrations related to radial basis networks.

Radial Basis Functions

Neuron Model

Here is a radial basis network with R inputs.

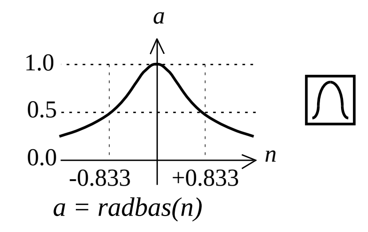


Notice that the expression for the net input of a `radbas` neuron is different from that of neurons in previous chapters. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The $\|$ `dist` $\|$ box in this figure accepts the input vector **p** and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is:

$$radbas(n) = e^{-n^2}$$

Here is a plot of the `radbas` transfer function.



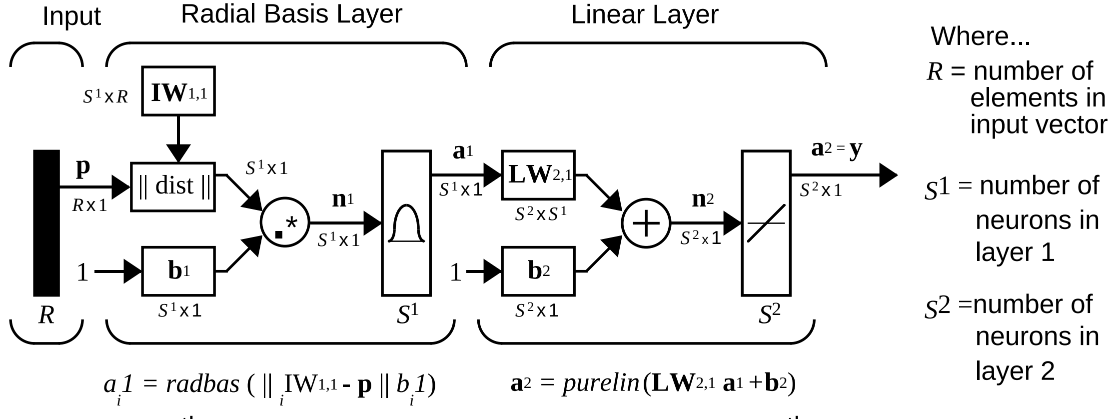
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **p**.

The bias b allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector **p** at vector distance of 8.326 (0.8326/b) from its weight vector **w**.

Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S^1 neurons, and an output linear layer of S^2 neurons.



The ‖ dist ‖ box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the distances between the input vector and vectors ${}_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector $\mathbf{b}^1$ and the output of ‖ dist ‖ are combined with the MATLAB operation .* , which does element-by-element multiplication.

The output of the first layer for a feed forward network *net* can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All of the details of designing this network are built into design functions `newrbe` and `newrb`, and their outputs can be obtained with `sim`.

We can understand how this network behaves by following an input vector **p** through the network to the output $\mathbf{a}^2$. If we present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1 its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0's (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now let us look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with `netprod`. Each neurons' output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input is `spread`, its net input is sqrt(-log(.5)) (or 0.8326), therefore its output is 0.5.

Exact Design (newrbe)

Radial basis networks can be designed with the function `newrbe`. This function can produce a network with zero error on training vectors. It is called in the following way.

```
net = newrbe(P,T,SPREAD)
```

The function `newrbe` takes matrices of input vectors `P` and target vectors `T`, and a spread constant `SPREAD` for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly `T` when the inputs are `P`.

This function `newrbe` creates as many `radbas` neurons as there are input vectors in P, and sets the first-layer weights to P'. Thus, we have a layer of `radbas` neurons in which each neuron acts as a detector for a different input vector. If there are Q input vectors, then there will be Q neurons.

Each bias in the first layer is set to 0.8326/`SPREAD`. This gives radial basis functions that cross 0.5 at weighted inputs of +/- `SPREAD`. This determines the width of an area in the input space to which each neuron responds. If `SPREAD` is 4, then each `radbas` neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. As we shall see, `SPREAD` should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights $IW^{2,1}$ (or in code, IW{2,1}) and biases b^2 (or in code, b{2}) are found by simulating the first-layer outputs a^1 (A{1}), and then solving the following linear expression.

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

We know the inputs to the second layer (A{1}) and the target (T), and the layer is linear. We can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

```
Wb = T/[P; ones(1,Q)]
```

Here `Wb` contains both weights and biases, with the biases in the last column. The sum-squared error will always be 0, as explained below.

We have a problem with C constraints (input/target pairs) and each neuron has C +1 variables (the C weights from the C `radbas` neurons, and a bias). A linear problem with C constraints and more than C variables has an infinite number of zero error solutions!

Thus, `newrbe` creates a network with zero error on training vectors. The only condition we have to meet is to make sure that `SPREAD` is large enough so that the active input regions of the `radbas` neurons overlap enough so that several `radbas` neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in the design. (However, `SPREAD` should not be so large that each neuron is effectively responding in the same, large, area of the input space.)

The drawback to `newrbe` is that it produces a network with as many hidden neurons as there are input vectors. For this reason, `newrbe` does not return an

acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

More Efficient Design (newrb)

The function newrb iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is:

```
net = newrb(P,T,GOAL,SPREAD)
```

The function newrb takes matrices of input and target vectors, P and T, and design parameters GOAL and, SPREAD, and returns the desired network.

The design method of newrb is similar to that of newrbe. The difference is that newrb creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most, is used to create a radbas neuron. The error of the new network is checked, and if low enough newrb is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met, or the maximum number of neurons is reached.

As with newrbe, it is important that the spread parameter be large enough that the radbas neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feed-forward network? Radial basis networks, even when designed efficiently with newrbe, tend to have many times more neurons than a comparable feed-forward network with tansig or logsig neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while radbas neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more radbas neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons being used, as can be seen in the next demonstration.

Demonstrations

The demonstration script `demorb1` shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Demonstration scripts `demorb3` and `demorb4` examine how the spread constant affects the design process for radial basis networks.

In `demorb3`, a radial basis network is designed to solve the same problem as in `demorb1`. However, this time the spread constant used is 0.01. Thus, each radial basis neuron returns 0.5 or lower, for any input vectors with a distance of 0.01 or more from its weight vector.

Because the training inputs occur at intervals of 0.1, no two radial basis neurons have a strong output for any given input.

In `demorb3`, it was demonstrated that having too small a spread constant can result in a solution that does not generalize from the input/target vectors used in the design. This demonstration, `demorb4`, shows the opposite problem. If the spread constant is large enough, the radial basis neurons will output large values (near 1.0) for all the inputs used to design the network.

If all the radial basis neurons always output 1, any information presented to the network becomes lost. No matter what the input, the second layer outputs 1's. The function `newrb` will attempt to find a network, but will not be able to do so because to numerical problems that arise in this situation.

The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

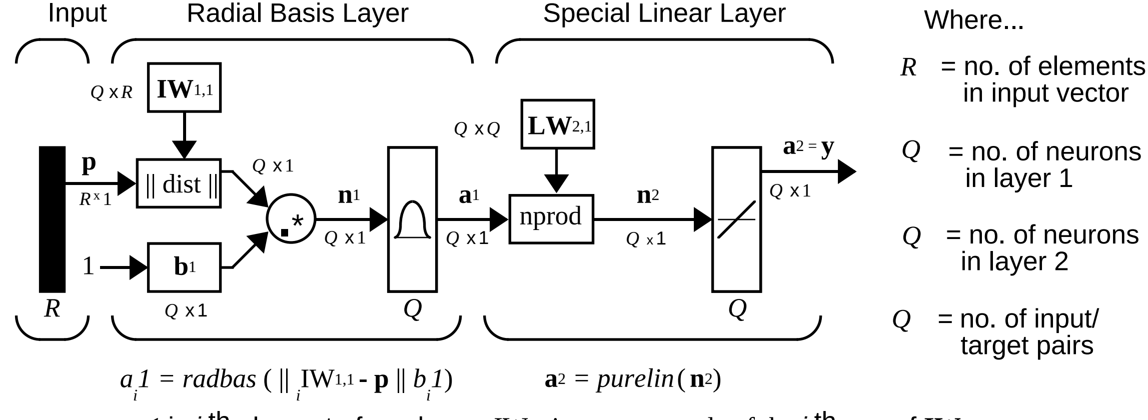
For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the left-most and right-most inputs.

Generalized Regression Networks

A generalized regression neural network (GRNN) is often used for function approximation. As discussed below, it has a radial basis layer and a special linear layer.

Network Architecture

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



a_i^1 is i th element of $\mathbf{a}^1$ where ${}_i\mathrm{IW}^{1,1}$ is a vector made of the i th row of $\mathbf{IW}^{1,1}$

Here the **nprod** box shown above (code function `normprod`) produces S^2 elements in vector $\mathbf{n}^2$. Each element is the dot product of a row of $\mathrm{LW}^{2,1}$ and the input vector $\mathbf{a}^1$, all normalized by the sum of the elements of $\mathbf{a}^1$. For instance, suppose that:

```
LW{1,2}= [1 -2;3 4;5 6];
a{1} = [7; -8;
```

Then

```
aout = normprod(LW{1,2},a{1})
aout =
    -23
     11
     13
```

The first layer is just like that for `newrbe` networks. It has as many neurons as there are input/ target vectors in **P**. Specifically, the first layer weights are set to **P**'. The bias $\mathbf{b}^1$ is set to a column vector of 0.8326/`SPREAD`. The user chooses `SPREAD`, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the `newbe` radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the product of its weighted input with its bias, calculated with `netprod`. Each neurons' output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input will be `spread`, and its net input will be sqrt(-log(.5)) (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here `LW{2,1}` is set to `T`.

Suppose we have an input vector **p** close to $\mathbf{p}_i$, one of the input vectors among the input vector/target pairs used in designing layer one weights. This input **p** produces a layer 1 $\mathbf{a}^i$ output close to 1. This leads to a layer 2 output close to $\mathbf{t}_i$, one of the targets used forming layer 2 weights.

A larger `spread` leads to a large area around the input vector where layer 1 neurons will respond with significant outputs.Therefore if `spread` is small the radial basis function is very steep so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network will tend to respond with the target vector associated with the nearest design input vector.

As `spread` gets larger the radial basis function's slope gets smoother and several neuron's may respond to an input vector. The network then acts like it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As `spread` gets larger more and more neurons contribute to the average with the result that the network function becomes smoother.

Design (newgrnn)

You can use the function `newgrnn` to create a GRNN. For instance, suppose that three input and three target vectors are defined as:

```
P = [4 5 6];
T = [1.5 3.6 6.7];
```

We can now obtain a GRNN with

```
net = newgrnn(P,T);
```

and simulate it with

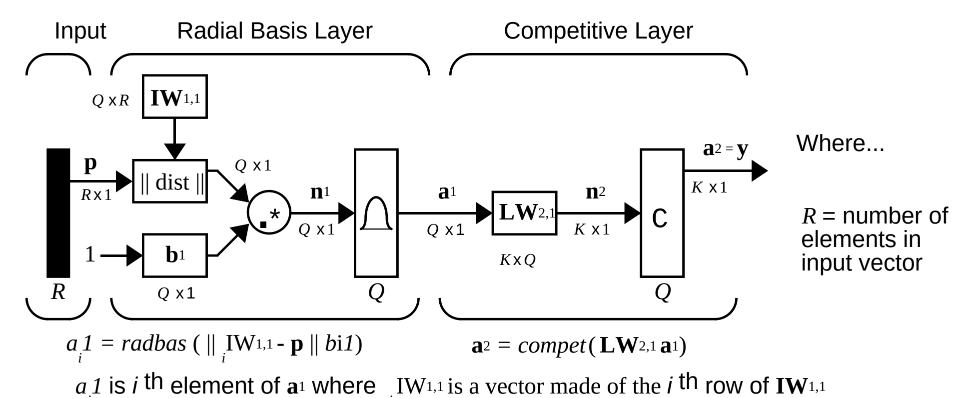
```
P = 4.5;
v = sim(net,P)
```

You might want to try `demogrn1`. It shows how to approximate a function with a GRNN.

Probabilistic Neural Networks

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors, and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these element is 1 and the rest is 0. Thus, each input vector is associated with one of K classes.

The first-layer input weights, $IW^{1,1}$ (`net.IW{1,1}`) are set to the transpose of the matrix formed from the Q training pairs, **P**'. When an input is presented the ||dist|| box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent the `radbas` transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector $\mathbf{a}^1$. If an input is close to several training vectors of a single class, it is represented by several elements of $\mathbf{a}^1$ that are close to 1.

The second-layer weights, $LW^{1,2}$ (`net.LW{2,1}`), are set to the matrix **T** of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0's elsewhere. (A function `ind2vec` is used to create the proper vectors.) The multiplication $\mathbf{T}\mathbf{a}^1$ sums the elements of $\mathbf{a}^1$ due to each of the K input classes. Finally, the second-layer transfer function, compete, produces a 1 corresponding to the largest element of $\mathbf{n}^2$, and 0's elsewhere. Thus, the network has classified the input vector into a specific one of K classes because that class had the maximum probability of being correct.

Design (newpnn)

You can use the function `newpnn` to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

```
P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]'
```

which yields

```
P =
     0     1     0     1     3     4     4
     0     1     3     4     1     1     3
Tc = [1 1 2 2 3 3 3];
```

which yields

```
Tc =
     1     1     2     2     3     3     3
```

We need a target matrix with 1's in the right place. We can get it with the function `ind2vec`. It gives a matrix with 0's except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
   (1,1)        1
   (1,2)        1
   (2,3)        1
   (2,4)        1
   (3,5)        1
   (3,6)        1
   (3,7)        1
```

Now we can create a network and simulate it, using the input P to make sure that it does produce the correct classifications. We use the function `vec2ind` to convert the output Y into a row Yc to make the classifications clear.

```
net = newpnn(P,T);
Y = sim(net,P)
Yc = vec2ind(Y)
```

Finally we get

```
Yc =
     1     1     2     2     3     3     3
```

We might try classifying vectors other than those that were used to design the network. We will try to classify the vectors shown below in P2.

```
P2 = [1 4;0 1;5 2]'

P2 =
     1     0     5
     4     1     2
```

Can you guess how these vectors will be classified? If we run the simulation and plot the vectors as we did before, we get

```
Yc =
     2     1     3
```

These results look good, for these test vectors were quite close to members of classes 2, 1 and 3 respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

You might want to try `demopnn1`. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

Summary

Radial basis networks can be designed very quickly in two different ways.

The first design method, `newrbe`, finds an exact solution. The function `newrbe` creates radial basis networks with as many radial basis neurons as there are input vectors in the training data.

The second method, `newrb`, finds the smallest network that can solve the problem within a given error goal. Typically, far fewer neurons are required by `newrb` than are returned `newrbe`. However, because the number of radial basis neurons is proportional to the size of the input space, and the complexity of the problem, radial basis networks can still be larger than backpropagation networks.

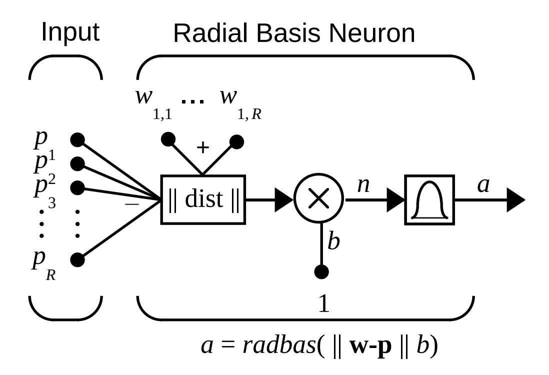
A generalized regression neural network (GRNN) is often used for function approximation. It has been shown that, given a sufficient number of hidden neurons, GRNNs can approximate a continuous function to an arbitrary accuracy.

Probabilistic neural networks (PNN) can be used for classification problems. Their design is straightforward and does not depend on training. A PNN is guaranteed to converge to a Bayesian classifier providing it is given enough training data. These networks generalize well.

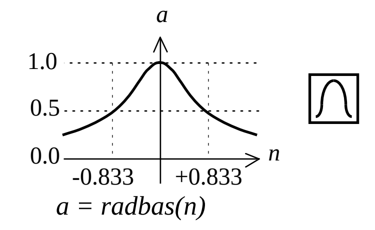
The GRNN and PNN have many advantages, but they both suffer from one major disadvantage. They are slower to operate because they use more computation than other kinds of networks to do their function approximation or classification.

Figures

Radial Basis Neuron

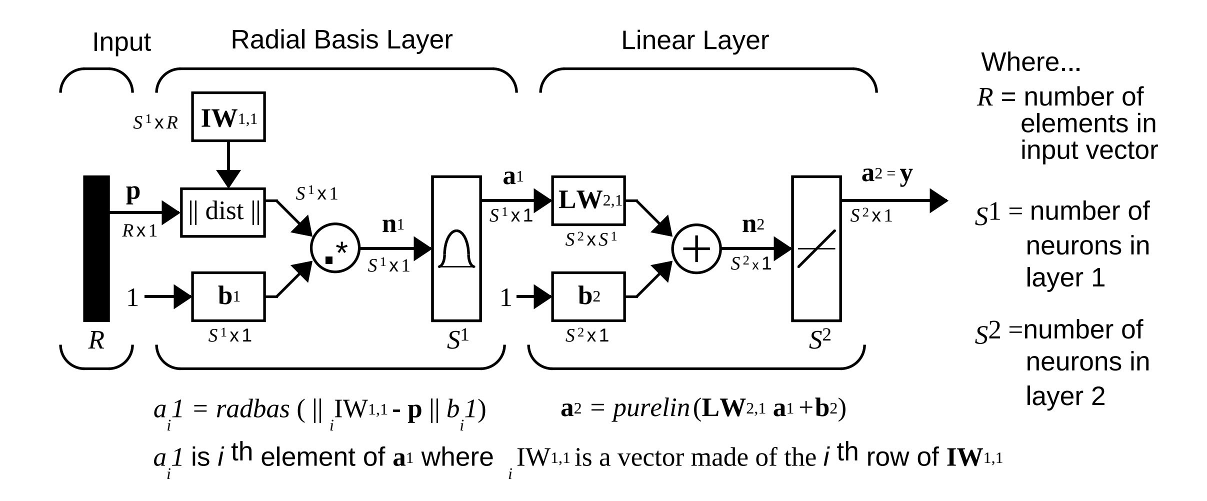


Radbas Transfer Function

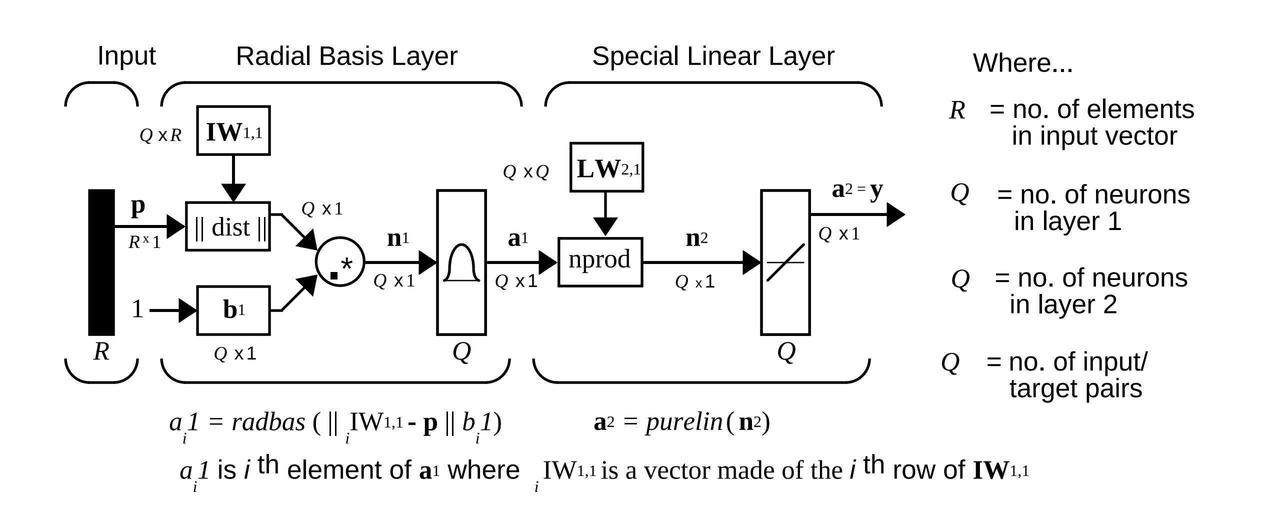


Radial Basis Function

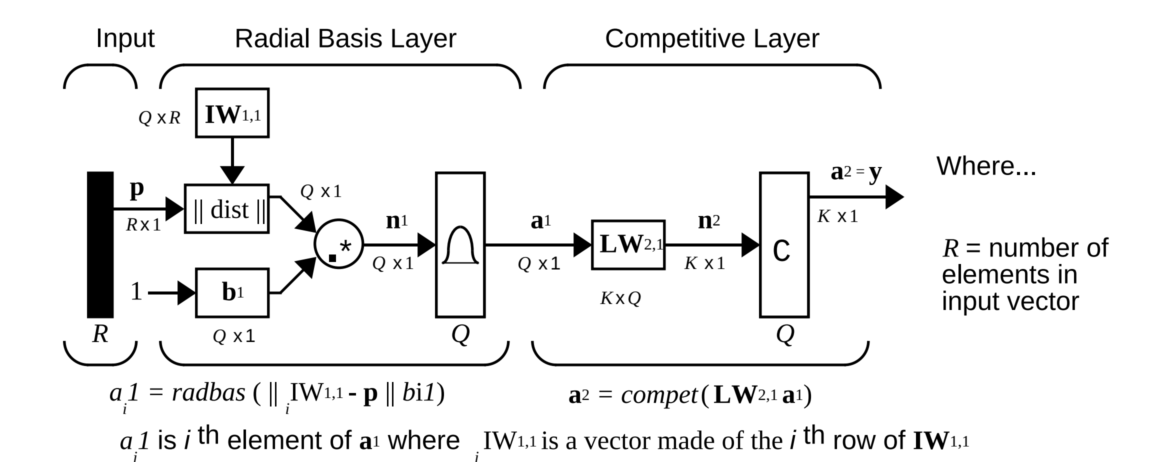
Radial Basis Network Architecture



Generalized Regression Neural Network Architecture



Probabilistic Neural Network Architecture



New Functions

This chapter introduces the following new functions.

Function	Description
`compet`	Competitive transfer function.
`dist`	Euclidean distance weight function
`dotprod`	Dot product weight function.
`ind2vec`	Convert indices to vectors.
`negdist`	Negative euclidean distance weight function
`netprod`	Product net input function.
`newgrnn`	Design a generalized regression neural network.

Function	Description
newpnn	Design a probabilistic neural network.
newrb	Design a radial basis network.
newrbe	Design an exact radial basis network.
normprod	Normalized dot product weight function.
radbas	Radial basis transfer function.
vec2ind	Convert vectors to indices.

8

Self-Organizing and Learn. Vector Quant. Nets

Introduction 8-2
Important Self-Organizing and LVQ Functions 8-2

Competitive Learning 8-3
Architecture 8-3
Creating a Competitive Neural Network (newc) 8-4
Kohonen Learning Rule (learnk) 8-5
Bias Learning Rule (learncon) 8-5
Training 8-6
Graphical Example 8-7

Self-Organizing Maps 8-9
Topologies (gridtop, hextop, randtop) 8-10
Distance Funct. (dist, linkdist, mandist, boxdist) 8-14
Architecture 8-17
Creating a Self Organizing MAP Neural Network (newsom) . 8-18
Training (learnsom) 8-19
Examples 8-23

Learning Vector Quantization Networks 8-31
Architecture 8-31
Creating an LVQ Network (newlvq) 8-32
LVQ1 Learning Rule(learnlv1) 8-35
Training 8-37
Supplemental LVQ2.1 Learning Rule (learnlv2) 8-38

Summary and Conclusions 8-40
Self-Organizing Maps 8-40
Learning Vector Quantizaton Networks 8-40
Figures 8-41
New Functions 8-42

Introduction

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors. A basic reference is

Kohonen, T. *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner. A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

LVQ networks, on the other hand, learn to classify input vectors into target classes chosen by the user.

You might consult the following reference:

Kohonen, T. *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

Important Self-Organizing and LVQ Functions

Competitive layers and self organizing maps can be created with `newc` and `newsom`, respectively. A listing of all self-organizing functions and demonstrations can be found by typing `help selforg`.

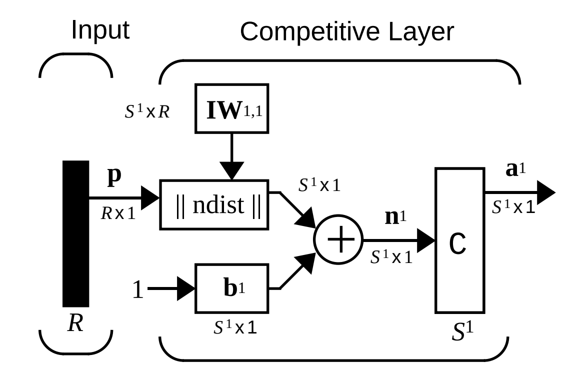
An LVQ network can be created with the function `newlvq`. For a list of all LVQ functions and demonstrations type `help lvq`.

Competitive Learning

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

Architecture

The architecture for a competitive network is shown below.



The ‖ dist ‖ box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S^1 elements. The elements are the negative of the distances between the input vector and vectors $_{i}\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The net input $\mathbf{n}^1$ of a competitive layer is computed by finding the negative distance between input vector **p** and the weight vectors and adding the biases **b**. If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input $\mathbf{n}^1$. The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in a later section on training.

Creating a Competitive Neural Network (newc)

A competitive neural network can be created with the function newc. We show how this works with a simple example.

Suppose we want to divide the following four two-element vectors into two classes.

```
p = [.1 .8  .1 .9; .2 .9 .1 .8]
p =
    0.1000    0.8000    0.1000    0.9000
    0.2000    0.9000    0.1000    0.8000
```

Thus, we have two vectors near the origin and two vectors near (1,1).

First, create a two-neuron layer with two input elements ranging from 0 to 1. The first argument gives the range of the two input vectors and the second argument says that there are to be two neurons.

```
net = newc([0 1; 0 1],2);
```

The weights are initialized to the center of the input ranges with the function midpoint. We can check to see these initial values as follows:

```
wts = net.IW{1,1}
wts =
    0.5000    0.5000
    0.5000    0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs, as we would expect when using midpoint for initialization.

The biases are computed by initcon, which gives

```
biases =
    5.4366
    5.4366
```

Now we have a network, but we need to train it to do the classification job.

Recall that each neuron competes to respond to an input vector **p**. If the biases are all 0, the neuron whose weight vector is closest to **p** gets the highest net input and, therefore, wins the competition and outputs 1. All other neurons output 0. We would like to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

Kohonen Learning Rule (learnk)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the ith neuron wins, the elements of the ith row of the input weight matrix are adjusted as shown below.

$$ {}_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1)) $$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

Bias Learning Rule (learncon)

One of the limitations of competitive networks is that some neurons may not always get allocated. In other words, some neuron weight vectors may start out far from any input vectors and never win the competition, no matter how long the training is continued. The result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this from happening, biases are used to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently

active neurons will get smaller, and biases of infrequently active neurons will get larger.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk`. Doing this helps make sure that the running average is accurate.

The result is that biases of neurons that haven't responded very frequently will increase versus biases of neurons that have responded frequently. As the biases of infrequently active neurons increase, the input space to which that neuron responds increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually the neuron will respond to an equal number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias will eventually get large enough so that it will be able to win. When this happens, it will move toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

Training

Now train the network for 500 epochs. Either `train` or `adapt` can be used.

```
net.trainParam.epochs = 500
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainr`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

This code produces

```
ans =
trainr
```

Thus, during each epoch, a single vector is chosen randomly and presented to the network and weight and bias values are updated accordingly.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

```
a = sim(net,p)
ac = vec2ind(a)
```

This yields

```
ac =
     1     2     1     2
```

We see that the network is trained to classify the input vectors into two groups, those near the origin, class 1, and those near (1,1), class 2.

It might be interesting to look at the final weights and biases. They are

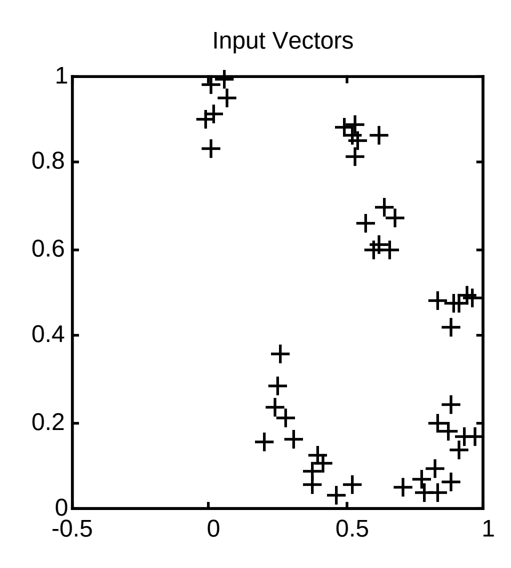
```
wts =
    0.8208    0.8263
    0.1348    0.1787
biases =
    5.3699
    5.5049
```

(You may get different answers if you run this problem, as a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to (1,1), while the vector formed from the second row of the weight matrix is close to the input vectors near the origin. Thus, the network has been trained, just by exposing it to the inputs, to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

Graphical Example

Competitive layers can be understood better when their weight vectors and input vectors are shown graphically. The diagram below shows 48 two-element input vectors represented as with '+' markers.



The input vectors above appear to fall into clusters. You can use a competitive network of eight neurons to classify the vectors into such clusters.

Try `democ1` to see a dynamic example of competitive learning.

Self-Organizing Maps

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The functions `gridtop`, `hextop` or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, `dist`, `boxdist`, `linkdist` and `mandist`. Link distance is the most common. These topology and distance functions are described in detail later in this section.

Here a self-organizing feature map network identifies a winning neuron i^* using the same procedure as employed by a competitive layer. However, instead of updating only the winning neuron, all neurons within a certain neighborhood $N_{i^*}(d)$ of the winning neuron are updated using the Kohonen rule. Specifically, we adjust all such neurons $i \in N_{i^*}(d)$ as follows.

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \text{ or} $$

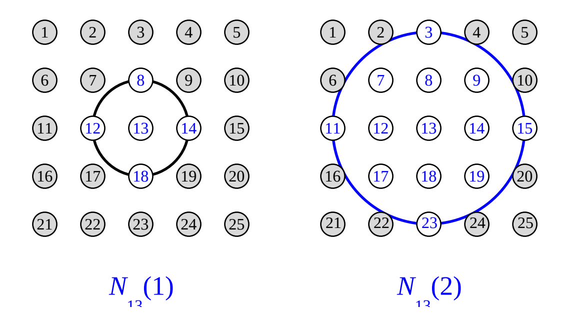
$$ {}_i\mathbf{w}(q) = (1-\alpha)\,{}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) $$

Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*.

$$ N_i(d) = \{j, d_{ij} \le d\} $$

Thus, when a vector $\mathbf{p}$ is presented, the weights of the winning neuron *and* its close neighbors move toward $\mathbf{p}$. Consequently, after many presentations, neighboring neurons will have learned vectors similar to each other.

To illustrate the concept of neighborhoods, consider the figure given below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.



These neighborhoods could be written as:

$$N_{13}(1) = \{8, 12, 13, 14, 18\} \text{ and}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$

Note that the neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or even three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line).You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

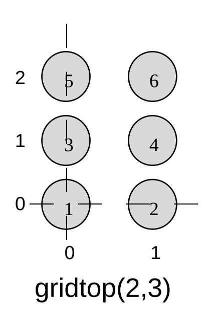
Topologies (gridtop, hextop, randtop)

You can specify different topologies for the original neuron locations with the functions `gridtop`, `hextop` or `randtop`.

The `gridtop` topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2 by 3 array of six neurons You can get this with:

```
pos = gridtop(2,3)
pos =
      0     1     0     1     0     1
      0     0     1     1     2     2
```

Here neuron 1 has the position (0,0); neuron 2 has the position (1,0); neuron 3 had the position (0,1); etc.



gridtop(2,3)

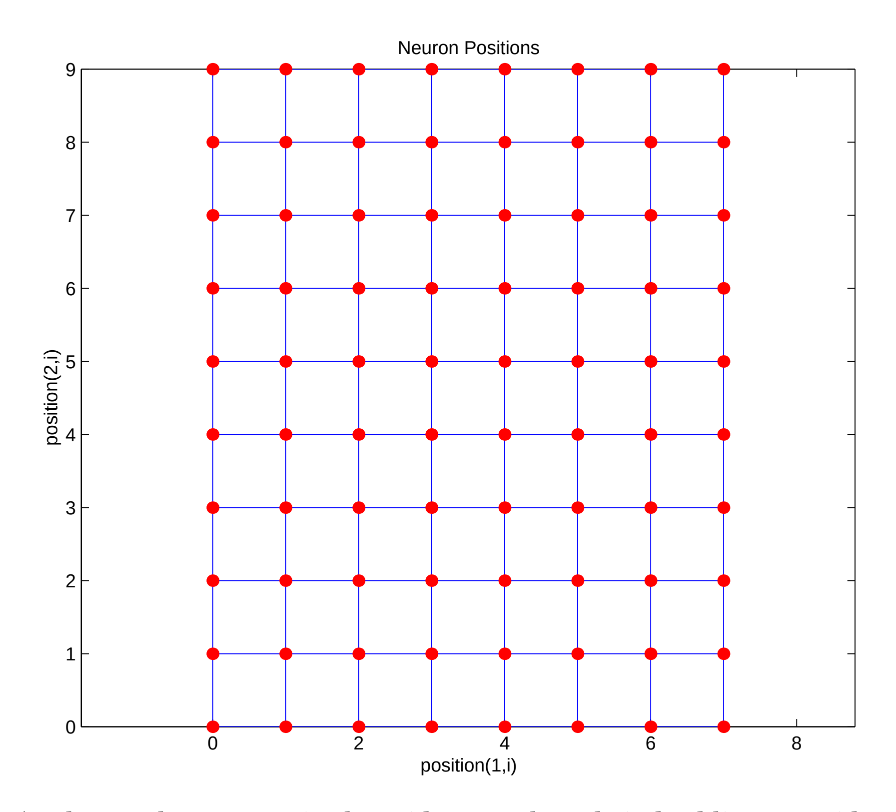
Note that had we asked for a `gridtop` with the arguments reversed we would have gotten a slightly different arrangement.

```
pos = gridtop(3,2)
pos =
     0     1     2     0     1     2
     0     0     0     1     1     1
```

An 8-by-10 set of neurons in a gridtop topology can be created and plotted with the code shown below

```
pos = gridtop(8,10);
plotsom(pos)
```

to give the following graph.



As shown, the neurons in the gridtop topology do indeed lie on a grid.

The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
pos = hextop(2,3)
pos =
              0    1.0000    0.5000    1.5000         0    1.0000
              0         0    0.8660    0.8660    1.7321    1.7321
```

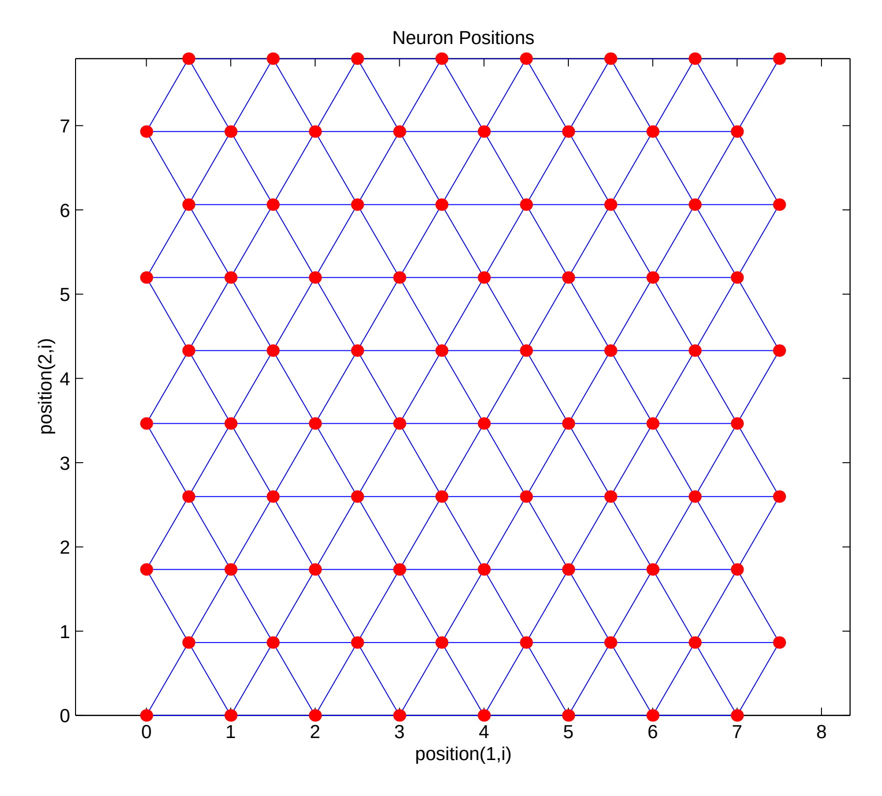
Note that `hextop` is the default pattern for SOFM networks generated with `newsom`.

An 8-by-10 set of neurons in a hextop topology can be created and plotted with the code shown below.

```
pos = hextop(8,10);
```

```
plotsom(pos)
```

to give the following graph.



Note the positions of the neurons in a hexagonal arrangement.

Finally, the `randtop` function creates neurons in an N dimensional random pattern. The following code generates a random pattern of neurons.

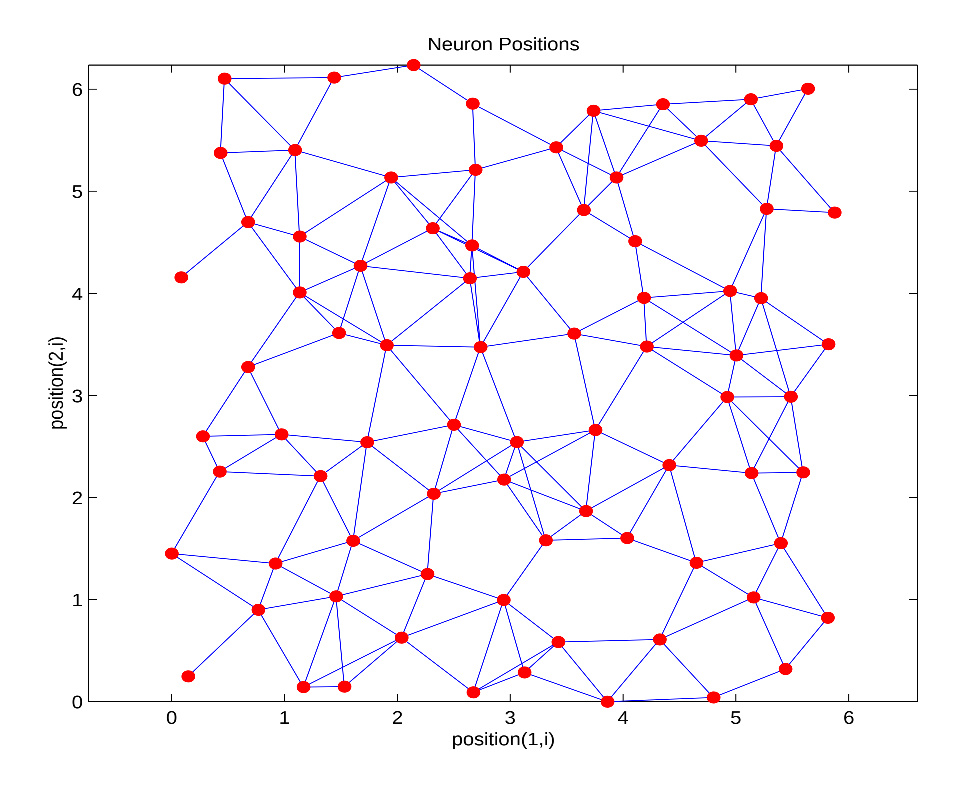
```
pos = randtop(2,3)
pos =
          0    0.7787    0.4390    1.0657    0.1470    0.9070
          0    0.1925    0.6476    0.9106    1.6490    1.4027
```

An 8-by-10 set of neurons in a randtop topology can be created and plotted with the code shown below

```
pos = randtop(8,10);
```

```
plotsom(pos)
```

to give the following graph.



For examples, see the help for these topology functions.

Distance Funct. (dist, linkdist, mandist, boxdist)

In this toolbox, there are four distinct ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

The `dist` function has been discussed before. It calculates the Euclidean distance from a *home* neuron to any other neuron. Suppose we have three neurons:

```
pos2 = [ 0 1 2; 0 1 2]
pos2 =
```

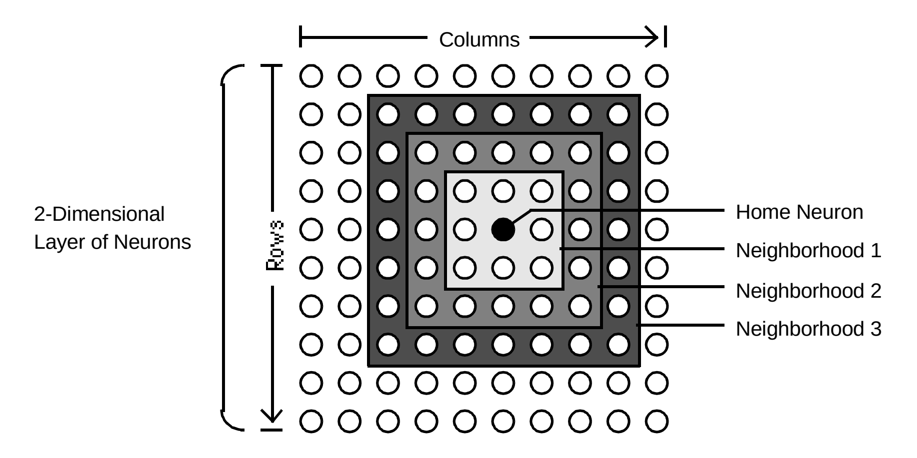
```
     0     1     2
     0     1     2
```

We find the distance from each neuron to the other with

```
D2 = dist(pos2)
D2 =
         0    1.4142    2.8284
    1.4142         0    1.4142
    2.8284    1.4142         0
```

Thus, the distance from neuron 1 to itself is 0, the distance from neuron 1 to neuron 2 is 1.414, etc. These are indeed the Euclidean distances as we know them.

The graph below shows a home neuron in a two-dimensional (`gridtop`) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1 includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the `dist` function, all the neighborhoods for an S neuron layer map are represented by an S-by-S matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function `boxdist`. Suppose that we have six neurons in a gridtop configuration.

```
pos = gridtop(2,3)
pos =
     0     1     0     1     0     1
     0     0     1     1     2     2
```

Then the box distances are

```
d = boxdist(pos)
d =
     0     1     1     1     2     2
     1     0     1     1     2     2
     1     1     0     1     1     1
     1     1     1     0     1     1
     2     2     1     1     0     1
     2     2     1     1     1     0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if we calculate the distances from the same set of neurons with `linkdist` we get

```
dlink =
     0     1     1     2     2     3
     1     0     2     1     3     2
     1     2     0     1     1     2
     2     1     1     0     2     1
     2     3     1     2     0     1
     3     2     2     1     1     0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

Thus if we have

```
W1 = [ 1 2; 3 4; 5 6]
W1 =
     1     2
     3     4
     5     6
```

and

```
P1= [1;1]
P1 =
      1
      1
```

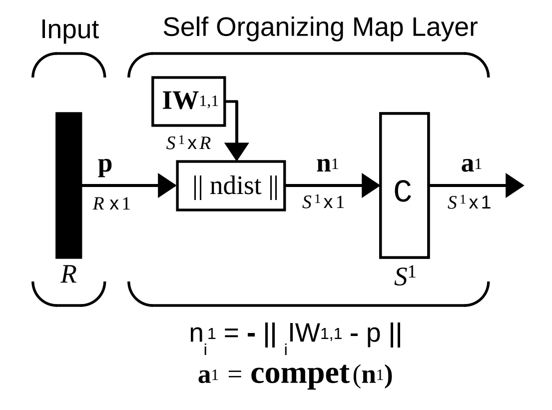
then we get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
      1
      5
      9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element $\mathbf{a}^1{}_i$ corresponding to $i*$,the winning neuron. All other output elements in $\mathbf{a}^1$ are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. As described previously, one can chose

from various topologies of neurons. Similarly, one can choose from various distance expressions to calculate neurons that are close to the winning neuron.

Creating a Self Organizing MAP Neural Network (newsom)

You can create a new SOFM network with the function `newsom`. This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that we want to create a network having input vectors with two elements that fall in the range 0 to 2 and 0 to 1 respectively. Further suppose that we want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is

```
net = newsom([0 2; 0 1] , [2 3]);
```

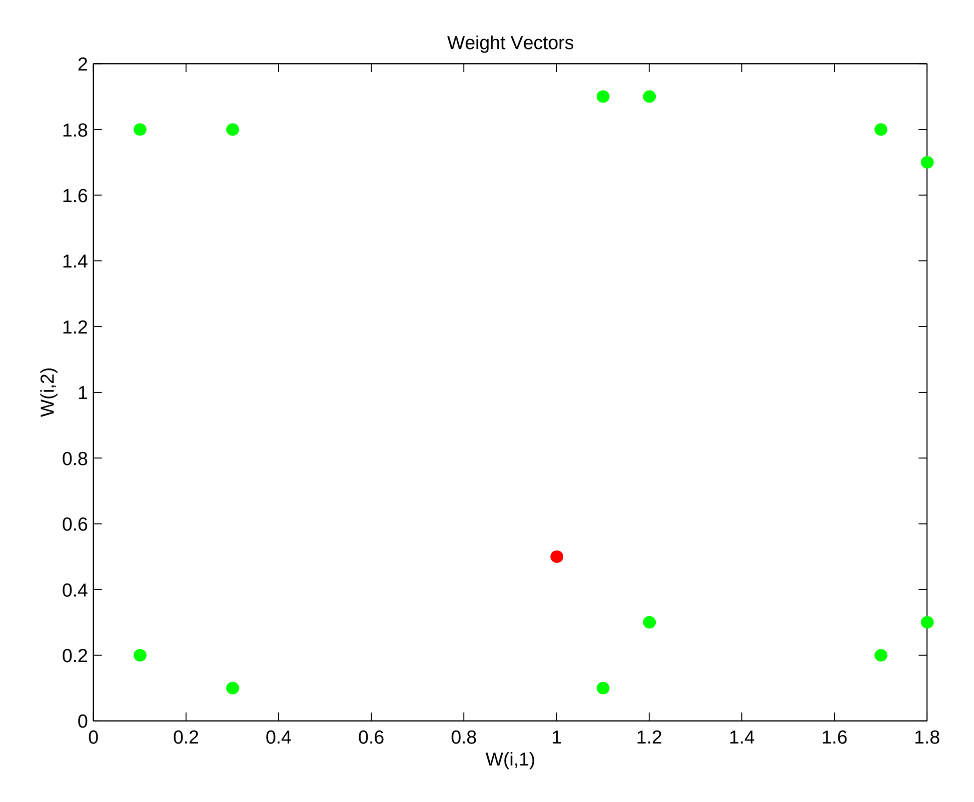
Suppose also that the vectors to train on are

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1 1.8 1.7;...
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9 1.7 1.8]
```

We can plot all of this with

```
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

to give



The various training vectors are seen as fuzzy gray spots around the perimeter of this figure. The initialization for `newsom` is `midpoint`. Thus, the initial network neurons are all concentrated at the black spot at (1, 0.5).

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated (`negdist`) to get the weighted inputs. The weighted inputs are also the net inputs (`netsum`). The net inputs compete (`compete`) so that only the neuron with the most positive net input will output a 1.

Training (learnsom)

Learning in a self-organizing feature map occurs for one vector at a time, independent of whether the network is trained directly (`trainr`) or whether it

is trained adaptively (`trains`). In either case, `learnsom` is the self-organizing map weight learning function.

First the network identifies the winning neuron. Then the weights of the winning neuron, and the other neurons in its neighborhood, are moved closer to the input vector at each learning step using the self-organizing map learning function `learnsom`. The winning neuron's weights are altered proportional to the learning rate. The weights of neurons in its neighborhood are altered proportional to half the learning rate. The learning rate and the neighborhood distance used to determine which neurons are in the winning neuron's neighborhood are altered during training through two phases.

Phase 1: Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts as the maximum distance between two neurons, and decreases to the tuning neighborhood distance. The learning rate starts at the ordering-phase learning rate and decreases until it reaches the tuning-phase learning rate. As the neighborhood distance and learning rate decrease over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

Phase 2: Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood distance stays at the tuning neighborhood distance, (which should include only close neighbors (i.e., typically 1.0). The learning rate continues to decrease from the tuning phase learning rate, but very slowly. The small neighborhood and slowly decreasing learning rate fine tune the network, while keeping the ordering learned in the previous phase stable. The number of epochs for the tuning part of training (or time steps for adaption) should be much larger than the number of steps in the ordering phase, because the tuning phase usually takes much longer.

Now let us take a look at some of the specific values commonly used in these networks.

Learning occurs according to the `learnsom` learning parameter, shown here with its default value.

`LP.order_lr`	`0.9`	Ordering-phase learning rate.
`LP.order_steps`	`1000`	Ordering-phase steps.
`LP.tune_lr`	`0.02`	Tuning-phase learning rate.
`LP.tune_nd`	`1`	Tuning-phase neighborhood distance.

`learnsom` calculates the weight change `dW` for a given neuron from the neuron's input `P`, activation `A2`, and learning rate `LR`:

```
dw =  lr*a2*(p'-w)
```

where the activation `A2` is found from the layer output `A` and neuron distances `D` and the current neighborhood size `ND`:

```
a2(i,q) = 1,   if a(i,q) = 1
        = 0.5, if a(j,q) = 1 and D(i,j) <= nd
        = 0,   otherwise
```

The learning rate `LR` and neighborhood size `NS` are altered through two phases: an ordering phase, and a tuning phase.

The ordering phase lasts as many steps as `LP.order_steps`. During this phase, `LR` is adjusted from `LP.order_lr` down to `LP.tune_lr`, and `ND` is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase `LR` decreases slowly from `LP.tune_lr` and `ND` is always set to `LP.tune_nd`. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered and the learning rate is slowly decreased over a

longer period to give the neurons time to spread out evenly across the input vectors.

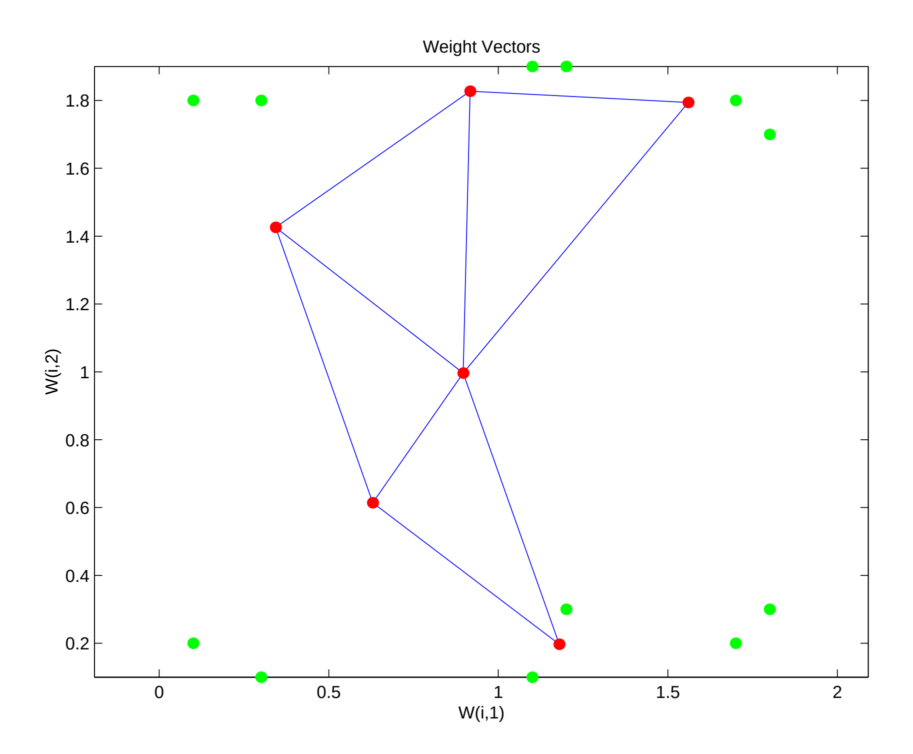
As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. Also, if input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

We can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;
net = train(net,P);
```

This training produces the following plot.



We can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

Examples

Two examples are described briefly below. You might try the demonstration scripts `demosm1` and `demosm2` to see similar examples.

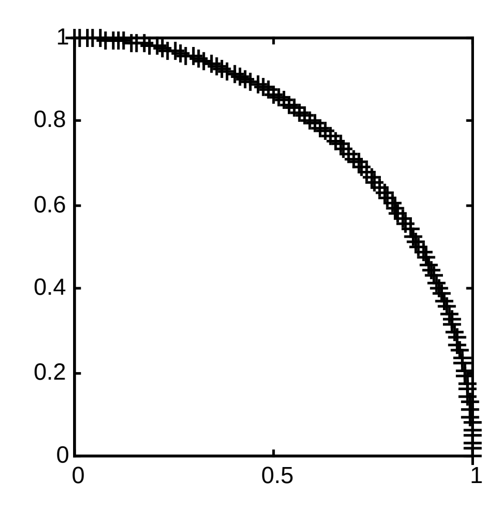
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

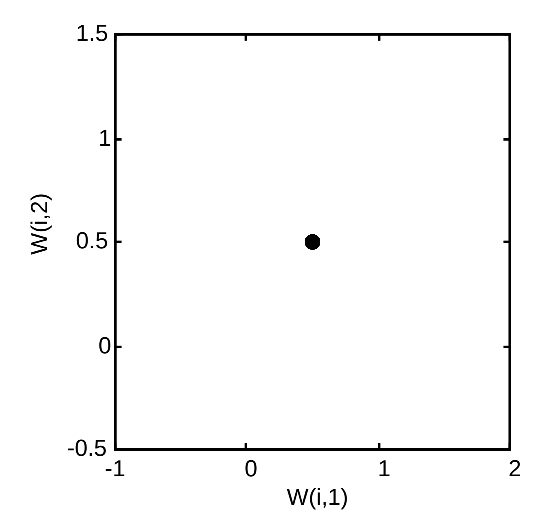
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

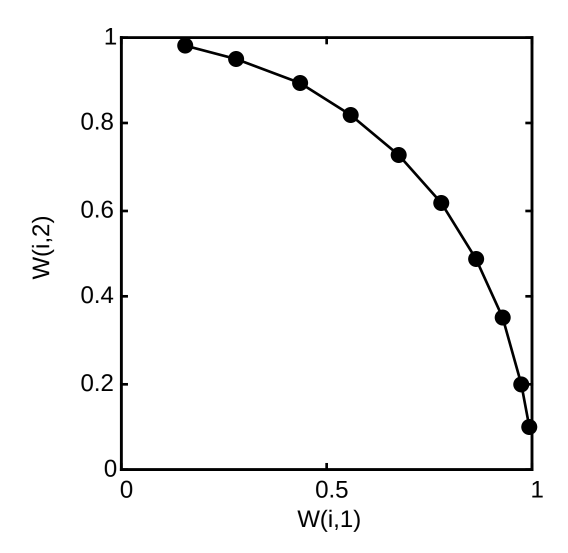


We define a a self-organizing map as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons will be at the center of the figure.



Of course, since all the weight vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial vectors does not guarantee identical training results.

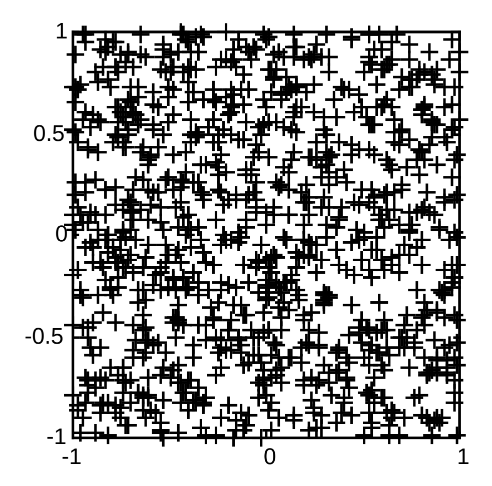
Two-Dimensional Self-Organizing Map

This example shows how a two-dimensional self-organizing map can be trained.

First some random input data is created with the following code.

```
P = rands(2,1000);
```

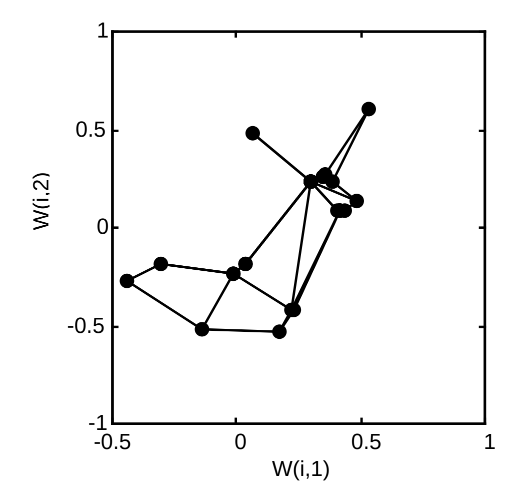
Here is a plot of these 1000 input vectors.



A two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with distances calculated according to the Manhattan distance neighborhood function `mandist`.

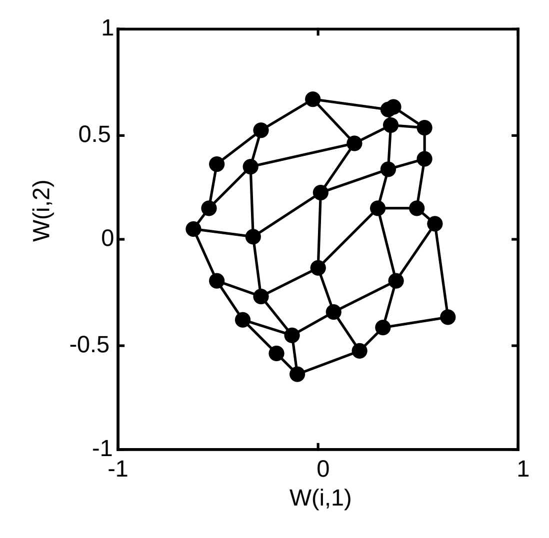
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



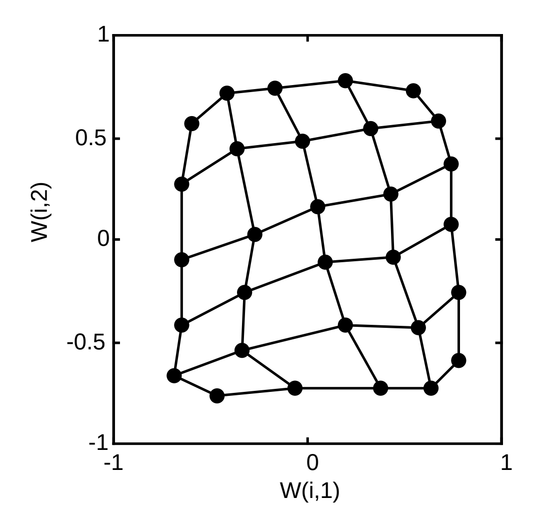
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

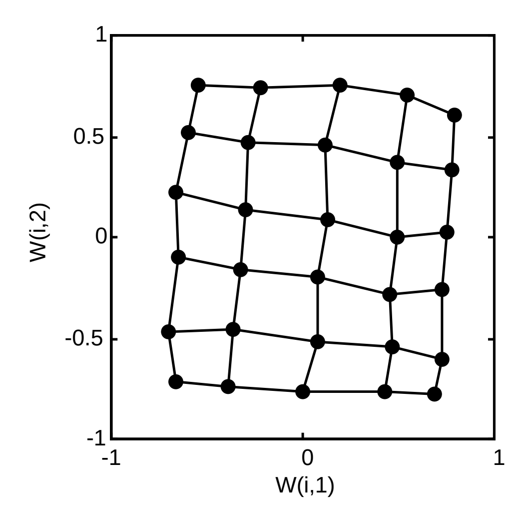


After 120 cycles, the map has begun to organize itself according to the topology of the input space which constrains input vectors.

The following plot, after 500 cycles, shows the map is more evenly distributed across the input space.



Finally, after 5000 cycles, the map is rather evenly spread across the input space. In addition, the neurons are very evenly spaced reflecting the even distribution of input vectors in this problem.



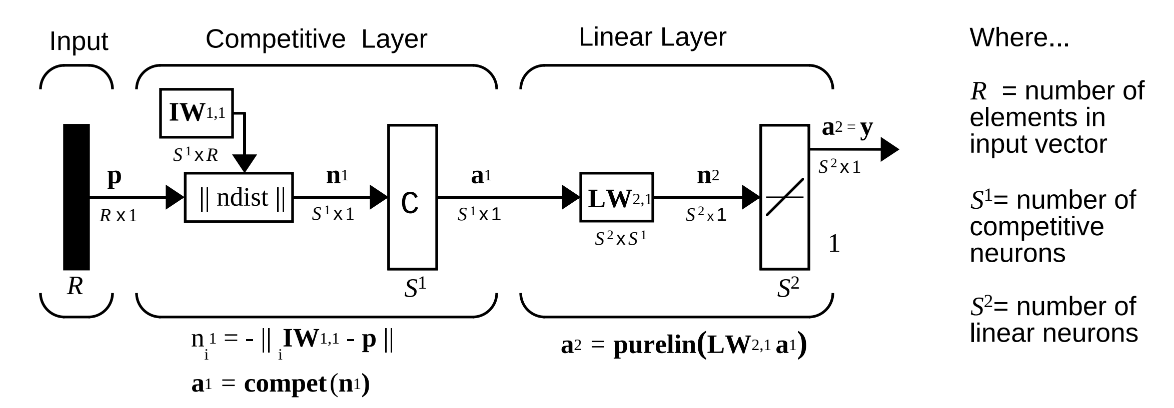
Thus a two-dimensional self-organizing map has learned the topology of its inputs' space.

It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

Learning Vector Quantization Networks

Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of Chapter 8. The linear layer transforms the competitive layer's classes into target classifications defined by the user. We refer to the classes learned by the competitive layer as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to S^1 subclasses. These, in turn, are combined by the linear layer to form S^2 target classes. (S^1 is always larger than S^2.)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that belongs to the linear layer target class No. 2. Then competitive neurons 1, 2, and 3, will have $\mathbf{LW}^{2,1}$ weights of 1.0 to neuron $\mathbf{n}^2$ in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1,2, and 3) win the competition and output a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a *1* in the ith row of $\mathbf{a}^1$ (the rest to the elements of $\mathbf{a}^1$ will be zero) effectively picks the ith column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1s from layer 1 get put into various classes, by the $\mathbf{LW}^{2,1}\mathbf{a}^1$ multiplication in layer 2.

We know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so we can specify the elements of $\mathbf{LW}^{2,1}$ at the start. However, we have to go through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. We discuss this training shortly. First consider how to create the original network.

Creating an LVQ Network (newlvq)

An LVQ network can be created with the function `newlvq`

```
net = newlvq(PR,S1,PC,LR,LF)
```

where:

- PR is an *R*-by-2 matrix of minimum and maximum values for *R* input elements.
- S^1 is the number of first layer hidden neurons.
- PC is an S^2 element vector of typical class percentages.
- LR is the learning rate (default 0.01).
- LF is the learning function (default is `learnlv1`).

Suppose we have 10 input vectors. We create a network that assigns each of these input vectors to one of four subclasses. Thus, we have four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2  0  0  0  0 +2 + 2 +3; ...
0 +1  -1 +2 +1  -1  -2 +1  -1   0]
```

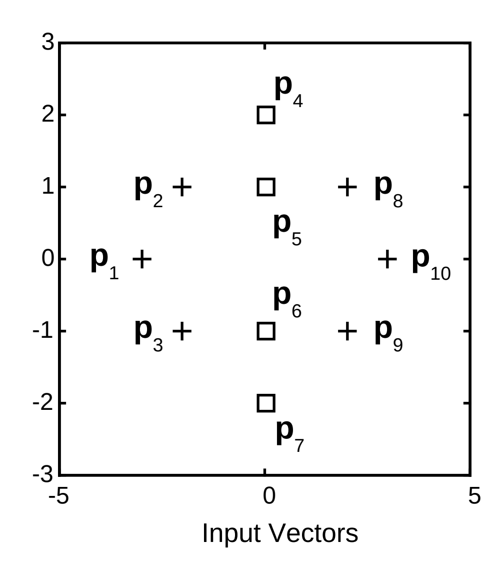
and

```
Tc = [1 1 1 2 2 2 2 1 1 1];
```

It may help to show the details of what we get from these two lines of code.

```
P =
    -3    -2    -2     0     0     0     0     2     2     3
     0     1    -1     2     1    -1    -2     1    -1     0
Tc =
     1     1     1     2     2     2     2     1     1     1
```

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. We want a network that classifies $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_8$, $\mathbf{p}_9$, and $\mathbf{p}_{10}$ to produce an output of 1, and that classifies vectors $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$ and $\mathbf{p}_7$ to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next we convert the `Tc` matrix to target vectors.

```
T = ind2vec(Tc)
```

This gives a sparse matrix `T` that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
```

```
     1     1     1     0     0     0     0     1     1     1
     0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if we have the first column of `P` as input, we should get the first column of `targets` as an output; and that output says the input falls in class 1, which is correct. Now we are ready to call `newlvq`.

We call `newlvq` with the proper arguments so that it creates a network with four neurons in the first layer and two neurons in the second layer. The first-layer weights are initialized to the center of the input ranges with the function `midpoint`. The second-layer weights have 60% (6 of the 10 in `Tc` above) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns will have a 1 in the second row (corresponding to class 2).

```
net = newlvq(minmax(P),4,[.6 .4], 0,1);
```

We can check to see the initial values of the first-layer weight matrix.

```
net.IW{1,1}
ans =
     0     0
     0     0
     0     0
     0     0
```

These zero weights are indeed the values at the midpoint of the range (-3 to +3) of the inputs, as we would expect when using `midpoint` for initialization.

We can look at the second-layer weights with

```
net.LW{2,1}
ans =
     1     1     0     0
     0     0     1     1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element. The input vector is classified as class 1; otherwise it is a class 2.

You may notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1), while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two

target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

We can simulate the network with `sim`. We use the original `P` matrix as input just to see what we get.

```
Y = sim(net,P);
Y = vec2ind(Yb4t)
Y =
      1     1     1     1     1     1     1     1     1     1
```

The network classifies all inputs into class 1. Since tis not what we want, we have to train the network (adjusting the weights of layer 1 only), before we can expect a good result. First we discuss two LVQ learning rules, and then we look at the training process.

LVQ1 Learning Rule(learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$$

Here we have input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector $\mathbf{p}$ is presented, and the distance from $\mathbf{p}$ to each row of the input weight matrix $\mathbf{IW}^{1,1}$ is computed with the function `ndist`. The hidden neurons of layer 1 compete. Suppose that the ith element of $\mathbf{n}^1$ is most positive, and neuron $i*$ wins the competition. Then the competitive transfer function produces a 1 as the $i*^{\text{th}}$ element of $\mathbf{a}^1$. All other elements of $\mathbf{a}^1$ are 0.

When $\mathbf{a}^1$ is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in $\mathbf{a}^1$ selects the class, k^* associated with the input. Thus, the network has assigned the input vector $\mathbf{p}$ to class k^* and $a^2_{k^*}$ will be 1. Of course, this assignment may be a good one or a bad one, for t_{k^*} may be 1 or 0, depending on whether the input belonged to class k^* or not.

We adjust the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector $\mathbf{p}$ if the assignment is correct, and to move the row away from $\mathbf{p}$ if the assignment is incorrect. So if $\mathbf{p}$ is classified correctly,

$$(a^2_{k^*} = t_{k^*} = 1)$$

we compute the new value of the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ as:

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1)) .$$

On the other hand, if $\mathbf{p}$ is classified incorrectly,

$$(a^2_{k^*} = 1 \neq t_{k^*} = 0) ,$$

we compute the new value of the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ as:

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

These corrections to the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ can be made automatically without affecting other rows of $\mathbf{IW}^{1,1}$ by backpropagating the output errors back to layer 1.

Such corrections move the hidden neuron towards vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is `learnlv1`. It can be applied during training.

Training

Next we need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. We do this with `train` as shown below. First set the training epochs to 150. Then, use `train`.

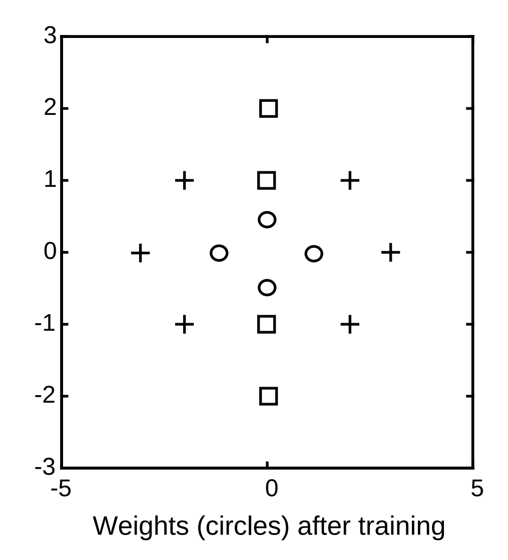
```
net.trainParam.epochs = 150;
```

```
net = train(net,P,T);
```

Now check on the first-layer weights.

```
net.IW{1,1}
ans =
    1.0927    0.0051
   -1.1028   -0.1288
         0   -0.5168
         0    0.3710
```

The following plot shows that these weights have moved toward their respective classification groups.



Weights (circles) after training

To check to see that these weights do indeed lead to the correct classification, take the matrix `P` as input and simulate the network. Then see what classifications are produced by the network.

```
Y = sim(net,P)
Yc = vec2ind(Y)
```

This gives

```
Yc =
```

```
     1     1     1     2     2     2     2     1     1     1
```

which is what we expected. As a last check, try an input close to a vector that was used in training.

```
pchk1 = [0; 0.5];
Y = sim(net,pchk1);
Yc1 = vec2ind(Y)
```

This gives

```
Yc1 =
     2
```

This looks right, for `pchk1` is close to other vectors classified as 2. Similarly,

```
pchk2 = [1; 0];
Y = sim(net,pchk2);
Yc2 = vec2ind(Y)
```

gives

```
Yc2 =
     1
```

This looks right too, for `pchk2` is close to other vectors classified as 1.

You might want to try the demonstration program `demolvq1`. It follows the discussion of training given above.

Supplemental LVQ2.1 Learning Rule (learnlv2)

The following learning rule is one that might be applied *after* first applying LVQ1. It may improve the result of the first learning. This particular version of LVQ2 (referred to as LVQ2.1 in the literature **[Koho97])** is embodied in the function `learnlv2`. Note again that LVQ2.1 is to be used only after LVQ1 has been applied

Learning here is similar to that in `learnlv1` except now two vectors of layer 1 that are closest to the input vector may be updated providing that one belongs to the correct class and one belongs to a wrong class and further providing that the input falls into a “window” near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s \quad \text{where} \quad s \equiv \frac{1-w}{1+w} \text{ ,}$$

(where d_i and d_j are the Euclidean distances of p from ${}_{i^*}\mathbf{IW}^{1,1}$ and ${}_{j^*}\mathbf{IW}^{1,1}$ respectively). We take a value for w in the range 0.2 to 0.3. If we pick, for instance, 0.25, then $s = 0.6$. This means that if the minimum of the two distance ratios is greater than 0.6, we adjust the two vectors. i.e., if the input is "near" the midplane, adjust the two vectors providing also that the input vector **p** and ${}_{j^*}\mathbf{IW}^{1,1}$ belong to the same class, and **p** and ${}_{i^*}\mathbf{IW}^{1,1}$ do not belong in the same class.

The adjustments made are

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1)) \quad \text{and}$$

$${}_{j^*}\mathbf{IW}^{1,1}(q) = {}_{j^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{j^*}\mathbf{IW}^{1,1}(q-1)) \text{ .}$$

Thus, given two vector closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors will be adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

Summary and Conclusions

Self-Organizing Maps

A competitive network learns to categorize the input vectors presented to it. If a neural network only needs to learn to categorize its input vectors, then a competitive network will do. Competitive networks also learn the distribution of inputs by dedicating more neurons to classifying parts of the input space with higher densities of input.

A self-organizing map learns to categorize input vectors. It also learns the distribution of input vectors. Feature maps allocate more neurons to recognize parts of the input space where many input vectors occur and allocate fewer neurons to parts of the input space where few input vectors occur.

Self-organizing maps also learn the topology of their input vectors. Neurons next to each other in the network learn to respond to similar vectors. The layer of neurons can be imagined to be a rubber net that is stretched over the regions in the input space where input vectors occur.

Self-organizing maps allow neurons that are neighbors to the winning neuron to output values. Thus the transition of output vectors is much smoother than that obtained with competitive layers, where only one neuron has an output at a time.

Learning Vector Quantizaton Networks

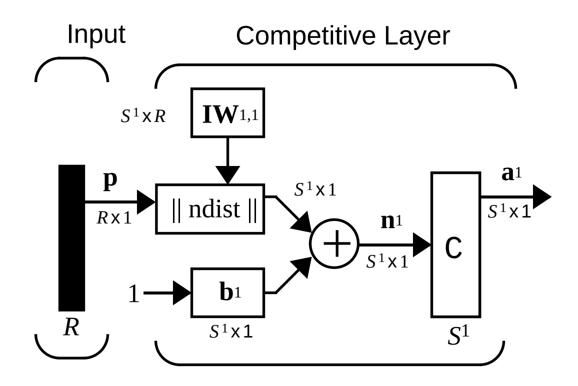
LVQ networks classify input vectors into target classes by using a competitive layer to find subclasses of input vectors, and then combining them into the target classes.

Unlike perceptrons, LVQ networks can classify any set of input vectors, not just linearly separable sets of input vectors. The only requirement is that the competitive layer must have enough neurons, and each class must be assigned enough competitive neurons.

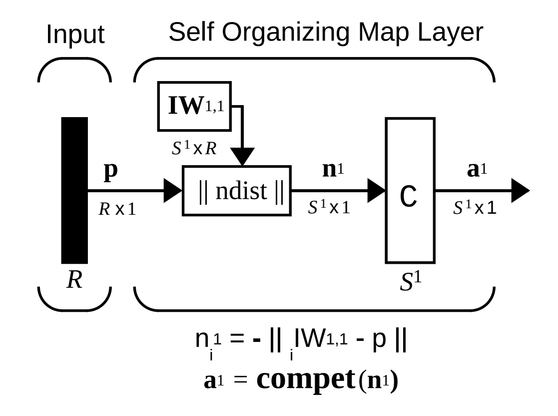
To ensure that each class is assigned an appropriate amount of competitive neurons, it is important that the target vectors used to initialize the LVQ network have the same distributions of targets as the training data the network is trained on. If this is done, target classes with more vectors will be the union of more subclasses.

Figures

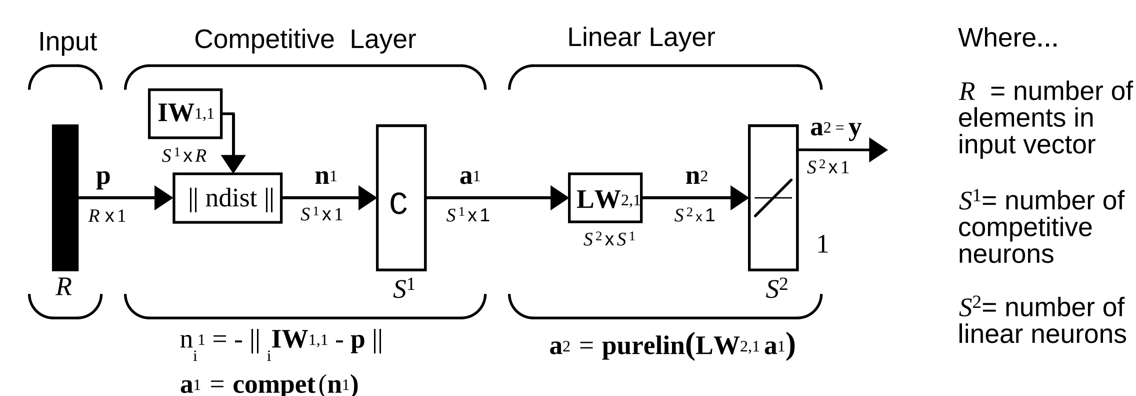
Competitive Network Architecture



Self Organizing Feature Map Architecture







New Functions

This chapter introduces the following new functions.

Function	Description
`newc`	Create a competitive layer.
`learnk`	Kohonen learning rule.
`newsom`	Create a self-organizing map.
`learncon`	Conscience bias learning function.
`boxdist`	Distance between two position vectors.
`dist`	Euclidean distance weight function.
`linkdist`	Link distance function.
`mandist`	Manhattan distance weight function.
`gridtop`	Gridtop layer topology function.
`hextop`	Hexagonal layer topology function.
`randtop`	Random layer topology function.

Function	Description
newlvq	Create a learning vector quantization network.
learnlv1	LVQ1 weight learning function.
learnlv2	LVQ2 weight learning function.

9

Recurrent Networks

Introduction 9-2
Important Recurrent Network Functions 9-2

Elman Networks 9-3
Architecture 9-3
Creating an Elman Network (newelm) 9-4
Training an Elman Network 9-5

Hopfield Network 9-8
Fundamentals 9-8
Architecture 9-8
Design (newhop) 9-10

Summary 9-15
Figures 9-16
New Functions 9-17

Introduction

Recurrent networks is a topic of considerable interest. This chapter covers two recurrent networks: Elman, and Hopfield networks.

Elman networks are two-layer backpropagation networks, with the addition of a feedback connection from the output of the hidden layer to its input. This feedback path allows Elman networks to learn to recognize and generate temporal patterns, as well as spatial patterns. The best paper on the Elman network is:

Elman, J. L., “Finding structure in time,” *Cognitive Science*, vol. 14, 1990, pp. 179-211.

The Hopfield network is used to store one or more stable target vectors. These stable vectors can be viewed as memories that the network recalls when provided with similar vectors that act as a cue to the network memory. You may want to pursue a basic paper in this field:

Li, J., A. N. Michel, and W. Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, November 1989, pp. 1405-1422.

Important Recurrent Network Functions

Elman networks can be created with the function `newelm`.

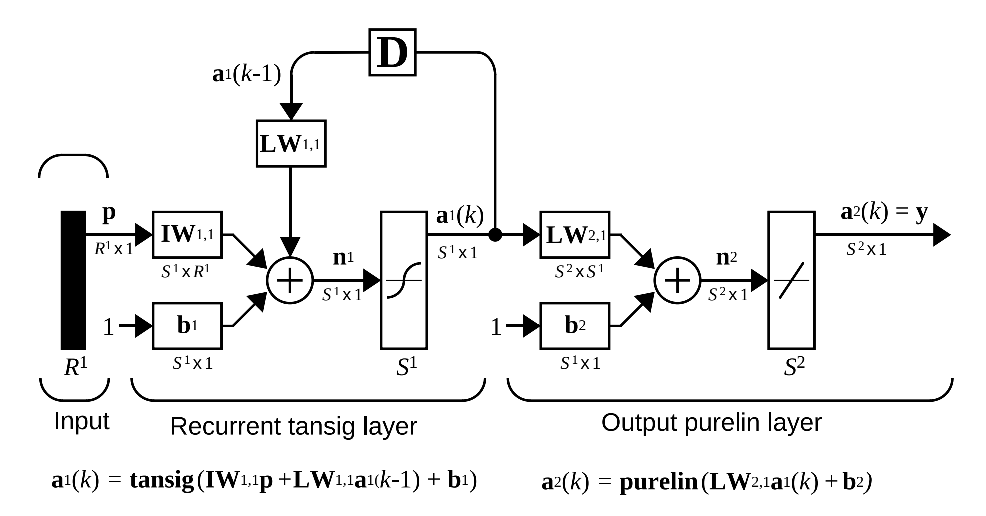
Hopfield networks can be created with the function `newhop`.

Type `help elman` or `help hopfield` to see a list of functions and demonstrations related to either of these networks.

Elman Networks

Architecture

The Elman network commonly is a two-layer network with feedback from the first-layer output to the first layer input. This recurrent connection allows the Elman network to both detect and generate time-varying patterns. A two-layer Elman network is shown below.



The Elman network has `tansig` neurons in its hidden (recurrent) layer, and `purelin` neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fit increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step.

Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different due to different feedback states.

Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

Creating an Elman Network (newelm)

An Elman network with two or more layers can be created with the function `newelm`. The hidden layers commonly have `tansig` transfer functions, so that is the default for `newelm`. As shown in the architecture diagram, purelin is commonly the output-layer transfer function.

The default backpropagation training function is `trainbfg`. One might use `trainlm`, but it tends to proceed so rapidly that it does not necessarily do well in the Elman network. The backprop weight/bias learning function default is `learngdm`, and the default performance function is `mse`.

When the network is created, each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method implemented in the function `initnw`.

Now consider an example. Suppose that we have a sequence of single-element input vectors in the range from 0 to 1. Suppose further that we want to have five hidden-layer tansig neurons and a single logsig output layer. The following code creates the desired network.

```
net = newelm([0 1],[5 1],{'tansig','logsig'});
```

Simulation

Suppose that we want to find the response of this network to an input sequence of eight digits that are either 0 or 1.

```
P = round(rand(1,8))
P =
     0     1     0     1     1     0     0     0
```

Recall that a sequence to be presented to a network is to be in cell array form. We can convert P to this form with

```
Pseq = con2seq(P)
Pseq =
    [0]    [1]    [0]    [1]    [1]    [0]    [0]    [0]
```

Now we can find the output of the network with the function `sim`.

```
Y = sim(net,Pseq)
Y =
Columns 1 through 5
   [1.9875e-04]   [0.1146]   [5.0677e-05]   [0.0017]   [0.9544]
Columns 6 through 8
    [0.0014]    [5.7241e-05]    [3.6413e-05]
```

We convert this back to concurrent form with

```
z = seq2con(Y);
```

and can finally display the output in concurrent form with

```
z{1,1}
ans =
  Columns 1 through 7
   0.0002   0.1146   0.0001   0.0017   0.9544   0.0014   0.0001
Column 8
    0.0000
```

Thus, once the network is created and the input specified, one need only call `sim`.

Training an Elman Network

Elman networks can be trained with either of two functions, `train` or `adapt`.

When using the function `train` to train an Elman network the following occurs.

At each epoch:

1. The entire input sequence is presented to the network, and its outputs are calculated and compared with the target sequence to generate an error sequence.

2. For each time step, the error is backpropagated to find *gradients* of errors for each weight and bias. This *gradient* is actually an approximation since the contributions of weights and biases to errors via the delayed recurrent connection are ignored.

3. This gradient is then used to update the weights with the backprop training function chosen by the user. The function `traingdx` is recommended.

When using the function adapt to train an Elman network, the following occurs.

At each time step:

1. Input vectors are presented to the network, and it generates an error.

2. The error is backpropagated to find gradients of errors for each weight and bias. This gradient is actually an approximation since the contributions of weights and biases to the error, via the delayed recurrent connection, are ignored.

3. This approximate gradient is then used to update the weights with the learning function chosen by the user. The function learngdm is recommended.

Elman networks are not as reliable as some other kinds of networks because both training and adaption happen using an approximation of the error gradient.

For an Elman to have the best chance at learning a problem it needs more hidden neurons in its hidden layer than are actually required for a solution by another method. While a solution may be available with fewer neurons, the Elman network is less able to find the most appropriate weights for hidden neurons since the error gradient is approximated. Therefore, having a fair number of neurons to begin with makes it more likely that the hidden neurons will start out dividing up the input space in useful ways.

The function train trains an Elman network to generate a sequence of target vectors when it is presented with a given sequence of input vectors. The input vectors and target vectors are passed to train as matrices P and T. Train takes these vectors and the initial weights and biases of the network, trains the network using backpropagation with momentum and an adaptive learning rate, and returns new weights and biases.

Let us continue with the example of the previous section, and suppose that we want to train a network with an input P and targets T as defined below

```
P = round(rand(1,8))
P =
     1     0     1     1     1     0     1     1
```

and

```
T = [0 (P(1:end-1)+P(2:end) == 2)]
T =
     0     0     0     1     1     0     0     1
```

Here T is defined to be 0, except when two 1's occur in P, in which case T is 1.

As noted previously, our network has five hidden neurons in the first layer.

```
net = newelm([0 1],[5 1],{'tansig','logsig'});
```

We use trainbfg as the training function and train for 100 epochs. After training we simulate the network with the input P and calculate the difference between the target output and the simulated network output.

```
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq);
z = seq2con(Y);
z{1,1};
diff1 = T - z{1,1}
```

Note that the difference between the target and the simulated output of the trained network is very small. Thus, the network is trained to produce the desired output sequence on presentation of the input vector **P**.

See Chapter 11 for an application of the Elman network to the detection of wave amplitudes.

Hopfield Network

Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points

The design method that we present is not perfect in that the designed network may have undesired spurious equilibrium points in addition to the desired ones. However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

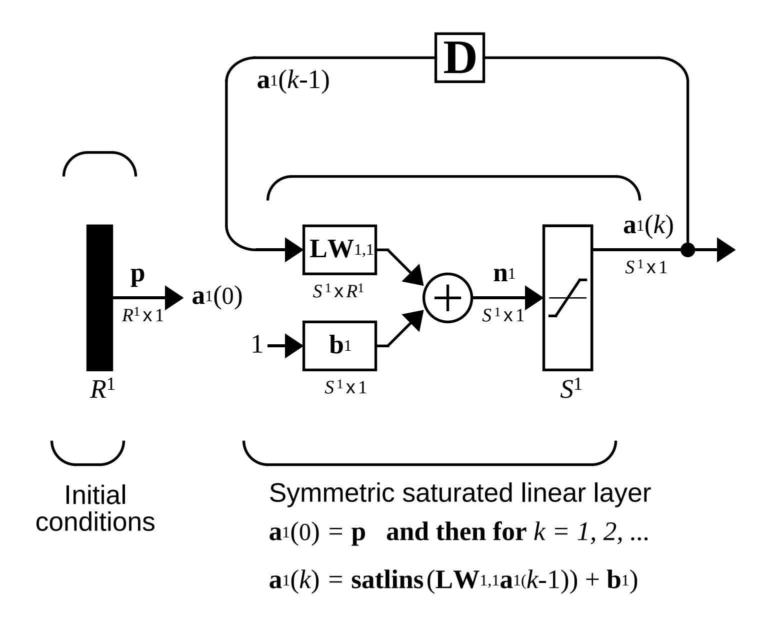
The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

The material in this section is based on the following paper: Jian-Hua Li, Anthony N. Michel and Wolfgang Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," IEEE Trans. on Circuits and Systems vol 36, no. 11, pp. 1405-22, November 1989.

For further information on Hopfield networks, read Chapter 18 of the *Hopfield Network* [HDB96].

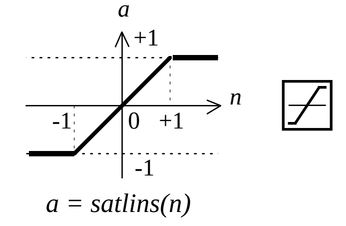
Architecture

The architecture of the network that we are using follows.



As noted, the *input* **p** to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function `satlins`.



Satlins Transfer Function

For inputs less than -1 `satlins` produces -1. For inputs in the range -1 to +1 it simply returns the input value. For inputs greater than +1 it produces +1.

This network can be tested with one or more input vectors which are presented as initial conditions to the network. After the initial conditions are given, the network produces an output which is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each

output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

Design (newhop)

Li et. al. [LiMi89] have studied a system that has the basic structure of the Hopfield network but is, in Li's own words, "easier to analyze, synthesize, and implement than the Hopfield model." The authors are enthusiastic about the reference article, as it has many excellent points and is one of the most readable in the field. However, the design is mathematically complex, and even a short justification of it would burden this guide. Thus, we present the Li design method, with thanks to Li et al., as a recipe that is found in the function `newhop`.

Given a set of target equilibrium points represented as a matrix **T** of vectors, `newhop` returns weights and biases for a recursive network. The network is guaranteed to have stable equilibrium points at the target vectors, but it could contain other spurious equilibrium points as well. The number of these undesired points is made as small as possible by the design method.

Once the network has been designed, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium points will find their targets. As suggested by the network figure, an array of input vectors may be presented at one time or in a batch. The network proceeds to give output vectors that are fed back as inputs. These output vectors can be can be compared to the target vectors to see how the solution is proceeding.

The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time. First you might check to see that the target equilibrium point vectors are indeed contained in the network. Then you could try other input vectors to determine the domains of attraction of the target equilibrium points and the locations of spurious equilibrium points if they are present.

Consider the following design example. Suppose that we want to design a network with two stable points in a three-dimensional space.

```
T = [-1 -1 1; 1 -1 1]'
T =
     -1     1
     -1    -1
      1     1
```

We can execute the design with

```
net = newhop(T);
```

Next we can check to make sure that the designed network is at these two points. We can do this as follows. (Since Hopfield networks have no inputs, the second argument to `sim` below is Q = 2 when using matrix notation).

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

This gives us

```
Y =
    -1     1
    -1    -1
     1     1
```

Thus, the network has indeed been designed to be stable at its design points. Next we can try another input condition that is not a design point, such as:

```
Ai = {[-0.9; -0.8; 0.7]}
```

This point is reasonably close to the first design point, so one might anticipate that the network would converge to that first point. To see if this happens, we run the following code. Note, incidentally, that we specified the original point in cell array form. This allows us to run the network for more than one step.

```
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

We get

```
Y =
    -1
    -1
     1
```

Thus, an original condition close to a design point did converge to that point.

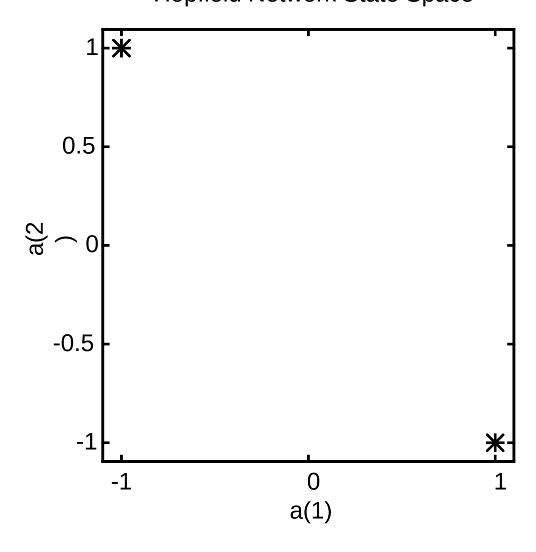
This is, of course, our hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include undesired spurious stable points that attract the solution.

Example

Consider a Hopfield network with just two neurons. Each neuron has a bias and weights to accommodate two-element input vectors weighted. We define the target equilibrium points to be stored in the network as the two columns of the matrix **T**.

```
T = [1 -1; -1 1]'
T =
      1     -1
     -1      1
```

Here is a plot of the Hopfield state space with the two stable points labeled with '*' markers.



These target stable points are given to `newhop` to obtain weights and biases of a Hopfield network.

```
net = newhop(T);
```

The design returns a set of weights and a bias for each neuron. The results are obtained from

```
W= net.LW{1,1}
```

which gives

```
W =
    0.6925   -0.4694
   -0.4694    0.6925
```

and from

```
b = net.b{1,1}
```

which gives

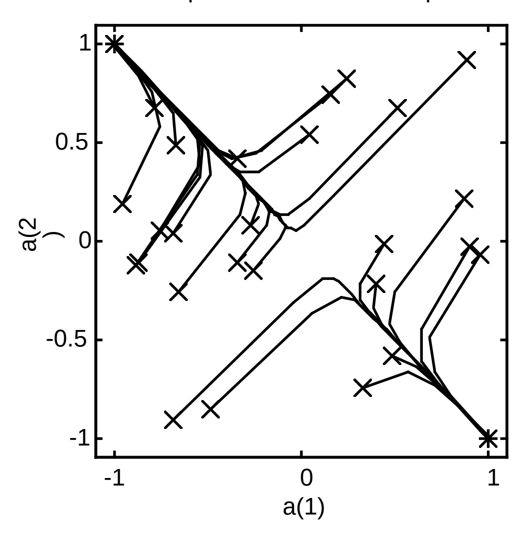
```
b =
   1.0e-16 *
    0.6900
    0.6900
```

Next the design is tested with the target vectors **T** to see if they are stored in the network. The targets are used as inputs for the simulation function `sim`.

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y =
     1    -1
    -1     1
```

As hoped, the new network outputs are the target vectors. The solution stays at its initial conditions after a single update and, therefore, will stay there for any number of updates.

Now you might wonder how the network performs with various random input vectors. Here is a plot showing the paths that the network took through its state space, to arrive at a target point.



This plot show the trajectories of the solution for various starting points. You can try the demonstration `demohop1` to see more of this kind of network behavior.

Hopfield networks can be designed for an arbitrary number of dimensions. You can try `demohop3` to see a three-dimensional design.

Unfortunately, Hopfield networks could have both unstable equilibrium points and spurious stable points. You can try demonstration programs `demohop2` and `demohop4` to investigate these issues.

Summary

Elman networks, by having an internal feedback loop, are capable of learning to detect and generate temporal patterns. This makes Elman networks useful in such areas as signal processing and prediction where time plays a dominant role.

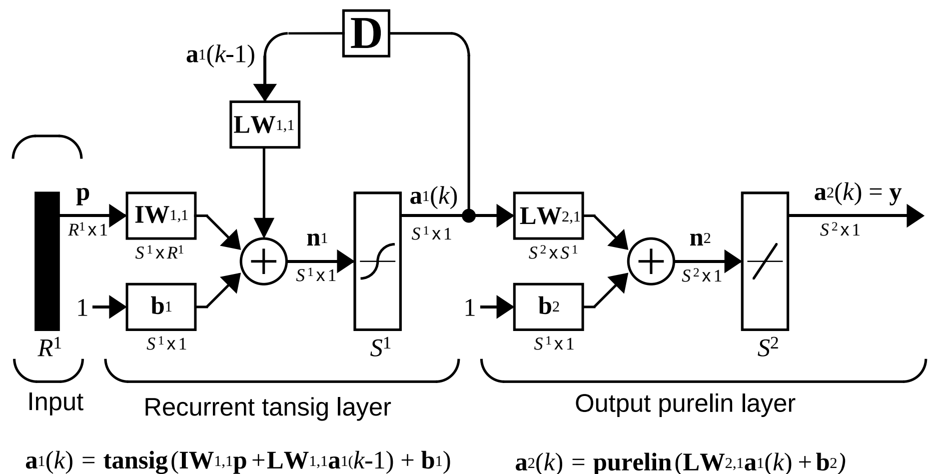
Because Elman networks are an extension of the two-layer sigmoid/linear architecture, they inherit the ability to fit any input/output function with a finite number of discontinuities. They are also able to fit temporal patterns, but may need many neurons in the recurrent layer to fit a complex function.

Hopfield networks can act as error correction or vector categorization networks. Input vectors are used as the initial conditions to the network, which recurrently updates until it reaches a stable output vector.

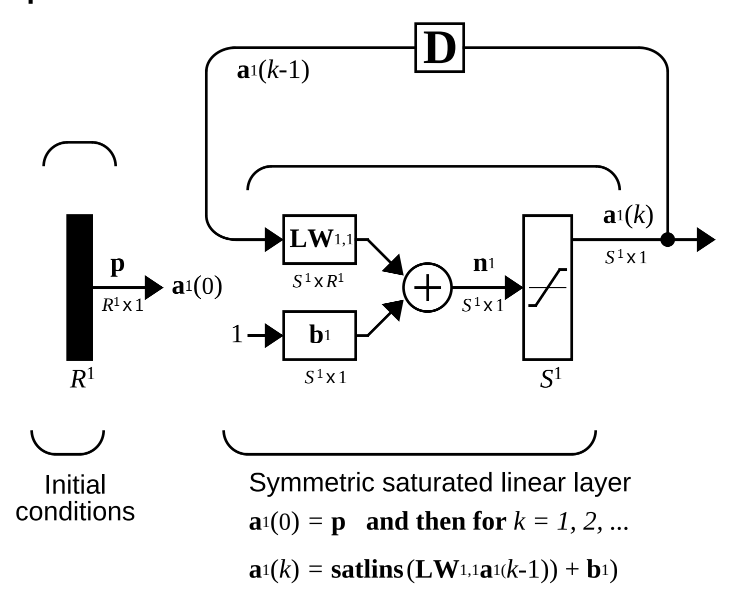
Hopfield networks are interesting from a theoretical standpoint, but are seldom used in practice. Even the best Hopfield designs may have spurious stable points that lead to incorrect answers. More efficient and reliable error correction techniques, such as backpropagation, are available.

Figures

Elman Network



Hopfield Network



New Functions

This chapter introduces the following new functions.

Function	Description
newelm	Create an Elman backpropagation network.
newhop	Create a Hopfield recurrent network.
satlins	Symmetric saturating linear transfer function.

10

Adaptive Filters and Adaptive Training

Introduction 10-2
Important Adaptive Functions 10-2

Linear Neuron Model 10-3

Adaptive Linear Network Architecture 10-4
Single ADALINE (newlin) 10-4

Mean Square Error 10-7

LMS Algorithm (learnwh) 10-8

Adaptive Filtering (adapt) 10-9
Tapped Delay Line 10-9
Adaptive Filter 10-9
Adaptive Filter Example 10-10
Prediction Example 10-13
Noise Cancellation Example 10-14
Multiple Neuron Adaptive Filters 10-16

Summary 10-18
Figures and Equations 10-18
New Functions 10-26

Introduction

The ADALINE (Adaptive Linear Neuron networks) networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can only solve linearly separable problems. However, here we will make use of the LMS (Least Mean Squares) learning rule, which is much more powerful than the perceptron learning rule. The LMS or Widrow-Hoff learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

In this chapter, we design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancellation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this subject is: Widrow B. and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall 1985.

We also consider the adaptive training of self organizing and competitive networks in this chapter.

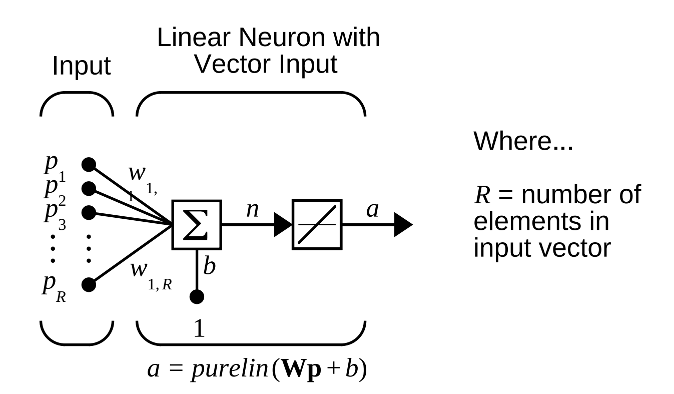
Important Adaptive Functions

This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

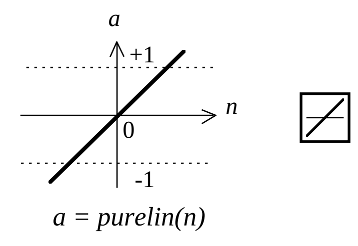
You can type `help linnet` to see a list of linear and adaptive network functions, demonstrations, and applications.

Linear Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, which we name `purelin`.



Linear Transfer Function

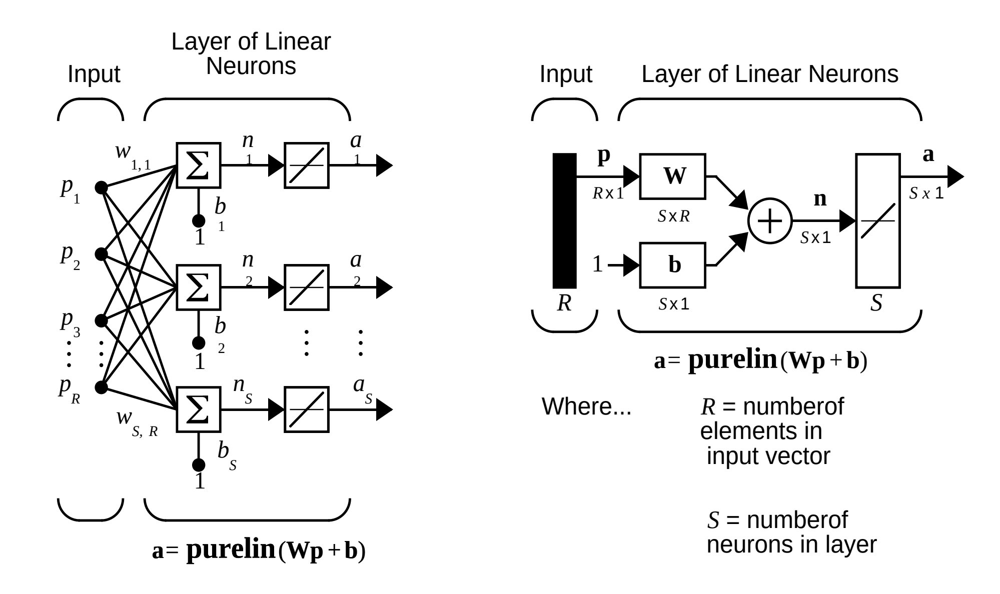
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights **W**.

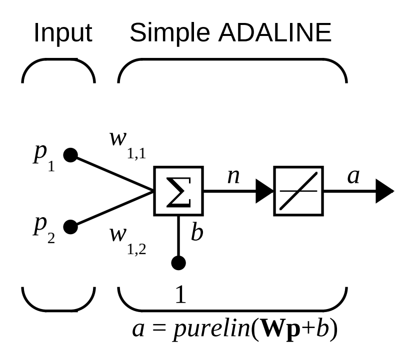


This network is sometimes called a MADALINE for Many ADALINES. Note that the figure on the right defines an S-length output vector **a**.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (newlin)

Consider a single ADALINE with two inputs. The diagram for this network is shown below.

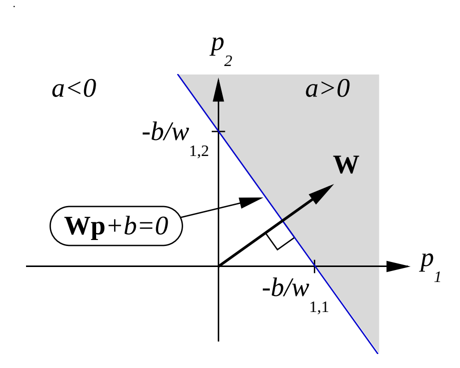


The weight matrix **W** in this case has only one row. The network output is:

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b \qquad \text{or}$$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary as shown below (adapted with thanks from [HDB96])



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories. Now you can find the network output with the function `sim`.

```
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE network with `newlin`, adjust its elements as you want and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. We want to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96].

LMS Algorithm (learnwh)

Adaptive networks will use the The LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown below, is discussed in detail in “Linear Filters” in Chapter 4.

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

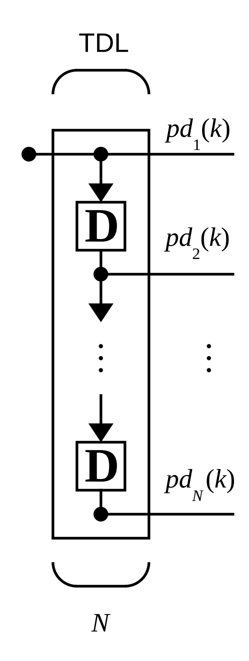
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k) .$$

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. Nevertheless, the ADALINE has been and is today one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

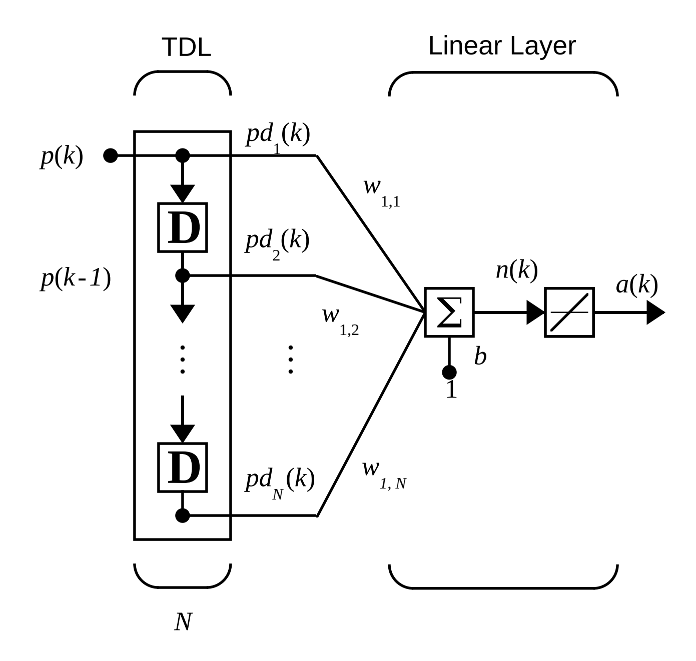
Tapped Delay Line

We need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown below. There the input signal enters from the left, and passes through N-1 delays. The output of the tapped delay line (TDL) is an N-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Adaptive Filter

We can combine a tapped delay line with an ADALINE network to create the adaptive filter shown below.



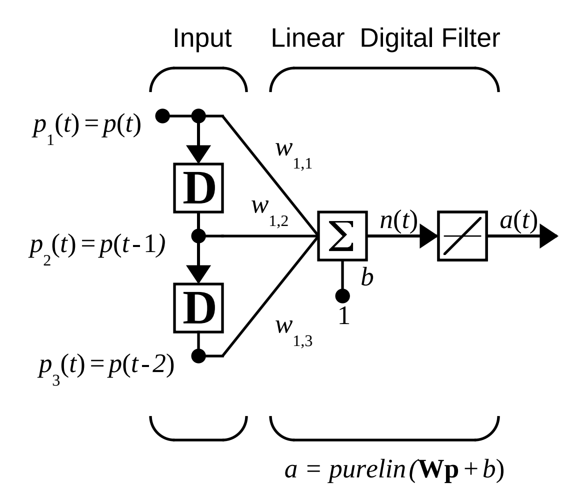
The output of the filter is given by

$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i}a(k - i + 1) + b$$

The network shown above is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85]. Let us take a look at the code that we use to generate and simulate such an adaptive network.

Adaptive Filter Example

First we will define a new linear network using `newlin`.



Assume that the input values have a range from 0 to 10. We can now define our single output network.

```
net = newlin([0,10],1);
```

We can specify the delays in the tapped delay line with

```
net.inputWeights{1,1}.delays = [0 1 2];
```

This says that the delay line is connected to the network weight matrix through delays of 0, 1, and 2 time units. (You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.)

We can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
net.b{1} = [0];
```

Finally we will define the initial values of the outputs of the delays as

```
pi ={1 2}
```

Note that these are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network. Now how about the input?

We assume that the input scalars arrive in a sequence, first the value 3, then the value 4, next the value 5, and finally the value 6. We can indicate this sequence by defining the values as elements of a cell array. (Note the curly brackets.)

```
p = {3 4 5 6}
```

Now we have a network and a sequence of inputs. We can simulate the network to see what its output is as a function of time.

```
[a,pf] = sim(net,p,pi);
```

This yields an output sequence

```
a =
    [46]    [70]    [94]    [118]
```

and final values for the delay outputs of

```
pf =
    [5]    [6].
```

The example is sufficiently simple that you can check it by hand to make sure that you understand the inputs, initial values of the delays, etc.

The network that we have defined can be trained with the function `adapt` to produce a particular output sequence. Suppose, for instance, we would like the network to produce the sequence of values 10, 20, 30, and 40.

```
T = {10 20 30 40}
```

We can train our defined network to do this, starting from the initial delay conditions that we used above. We specify 10 passes through the input sequence with

```
net.adaptParam.passes = 10;
```

Then we can do the training with

```
[net,y,E pf,af] = adapt(net,p,T,pi);
```

This code returns the final weights, bias, and output sequence shown below.

```
wts = net.IW{1,1}
wts =
    0.5059    3.1053    5.7046
```

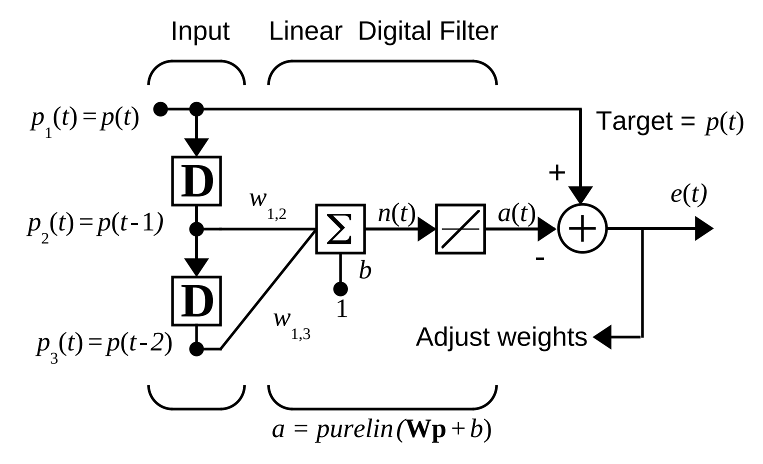
```
bias = net.b{1}
bias =
   -1.5993
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

Presumably, if we ran for additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with `adapt`. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as or prediction or noise cancellation.

Prediction Example

Suppose that we want to use an adaptive filter to predict the next value of a stationary random process, *p(t)*. We use the network shown below to do this.



Predictive Filter: $a(t)$ is approximation to $p(t)$

The signal to be predicted, *p(t),* enters from the left into a tapped delay line. The previous two values of *p(t)* are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error *e(t)* on the far right. If this error is zero, then the network output *a(t)* is exactly equal to *p(t)*, and the network has done its prediction properly.

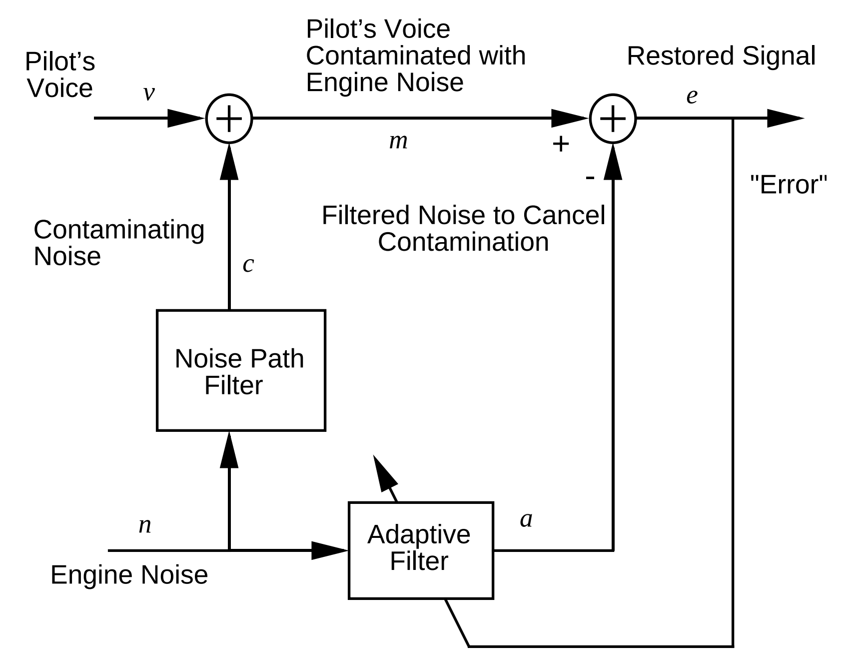
A detailed analysis of this network is not appropriate here, but we can state the main points. Given the autocorrelation function of the stationary random process *p(t)*, the error surface, the maximum learning rate, and the optimum values of the weights can be calculated. Commonly, of course, one does not have detailed information about the random process, so these calculations cannot be performed. But this lack does not matter to the network. The network, once initialized and operating, adapts at each time step to minimize the error and in a relatively short time is able to predict the input p(t).

Chapter 10 of [HDB96] presents this problem, goes through the analysis, and shows the weight trajectory during training. The network finds the optimum weights on its own without any difficulty whatsoever.

You also can try demonstration program `nnd10nc` to see an adaptive noise cancellation program example in action. This demonstration allows you to pick a learning rate and *momentum*, (see Chapter 5), and shows the learning trajectory, and the original and cancellation signals verses time.

Noise Cancellation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit is added to the voice signal, and the resultant signal heard by passengers would be of low quality. We would like to obtain a signal that contains the pilot's voice, but not the engine noise. We can do this with an adaptive filter if we obtain a sample of the engine noise and apply it as the input to the adaptive filter.



Adaptive Filter Adjusts to Minimize Error.
This removes the engine noise from contaminated signal, leaving the pilot's voice as the "error."

Here we adaptively train the neural linear network to predict the combined pilot/engine signal m from an engine signal n. Notice that the engine signal n does not tell the adaptive network anything about the pilot's voice signal contained in m. However, the engine signal n. does give the network information it can use to predict the engine's contribution to the pilot/engine signal m.

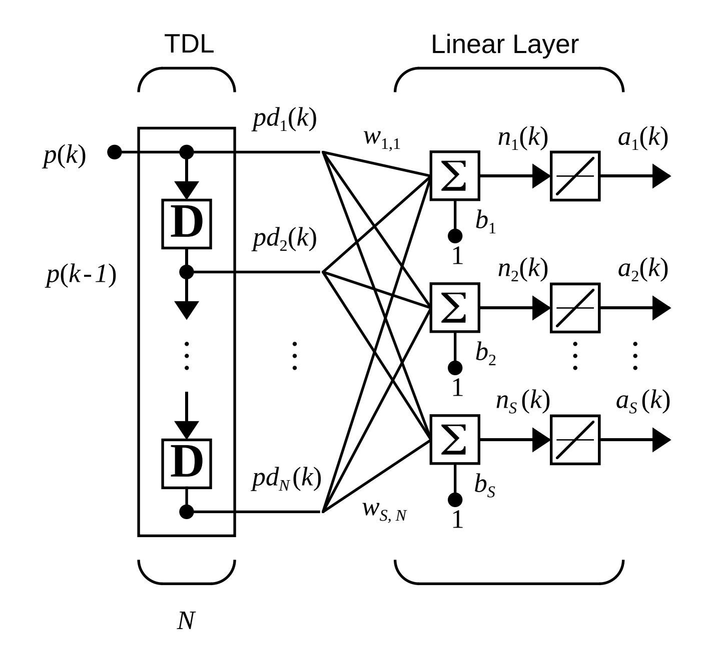
The network will do its best to adaptively output m. In this case, the network can only predict the engine interference noise in the pilot/engine signal m. The network error e is equal to m, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, e contains only the pilot's voice! Our linear adaptive network adaptively learns to cancel the engine noise.

Note, in closing, that such adaptive noise canceling generally does a better job than a classical filter because the noise here is subtracted from rather than filtered out of the signal m.

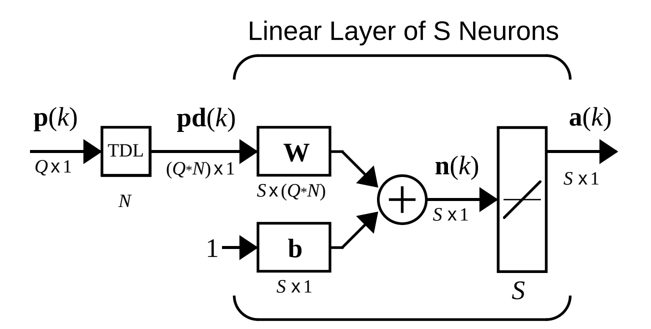
Try `demolin8` for an example of adaptive noise cancellation.

Multiple Neuron Adaptive Filters

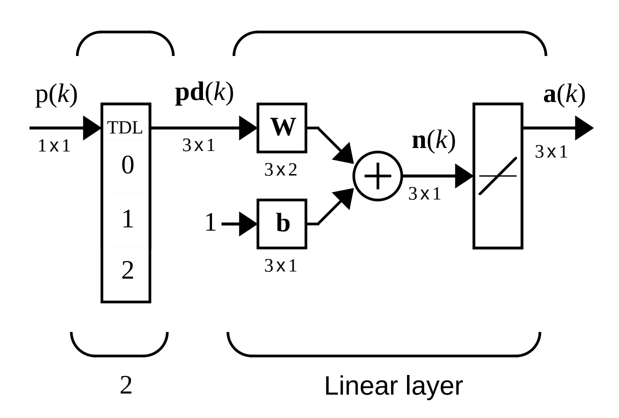
We may want to use more than one neuron in an adaptive system, so we need some additional notation. A tapped delay line can be used with S linear neurons as shown below.



Alternatively, we can show this same network in abbreviated form.



If we want to show more of the detail of the tapped delay line and there are not too many delays, we can use the following notation.



Here we have a tapped delay line that sends the current signal, the previous signal, and the signal delayed before that to the weight matrix. We could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays are shown in increasing order as they go from top to bottom.

Summary

The ADALINE (Adaptive Linear Neuron networks) networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. They make use of the LMS (Least Mean Squares) learning rule, which is much more powerful that the perceptron learning rule. The LMS or Widrow-Hoff learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

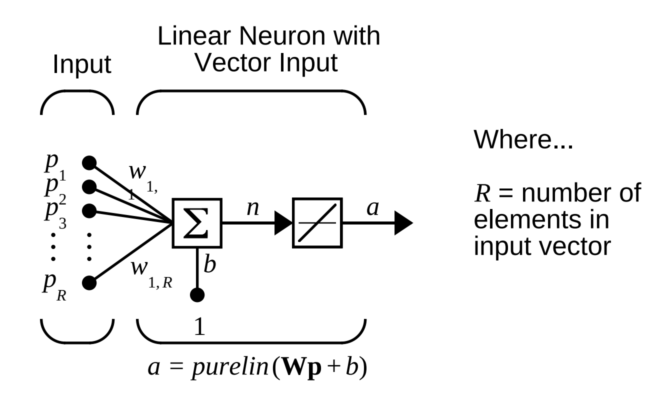
In this chapter, we design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors.

Adaptive linear filters have many practical applications such as noise cancellation, signal processing, and prediction in control and communication systems.

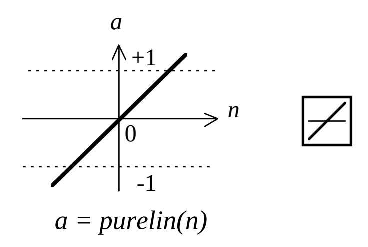
This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

Figures and Equations

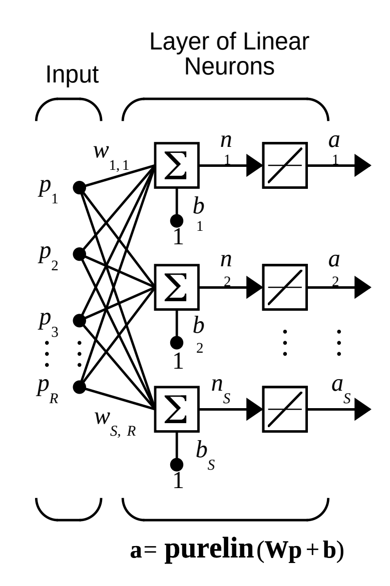
Linear Neuron

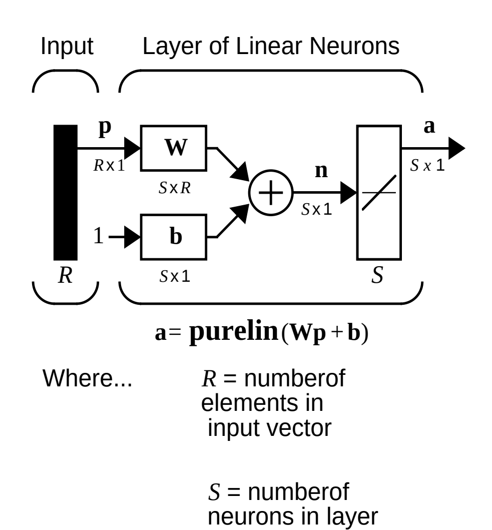


Purelin Transfer Function



MADALINE



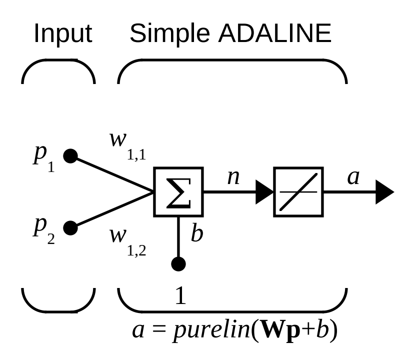


Where...

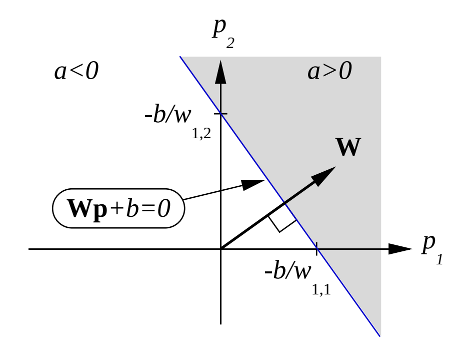
R = numberof elements in input vector

S = numberof neurons in layer

ADALINE



Decision Boundary



Mean Square Error

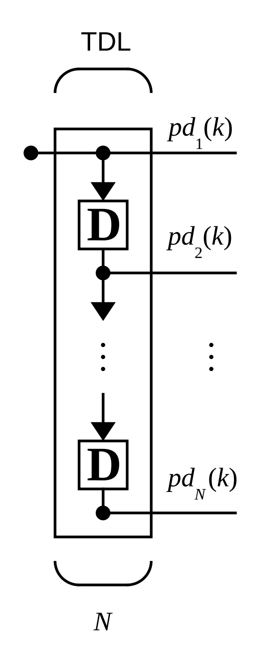
$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

LMS (Widrow-Hoff) Algorithm

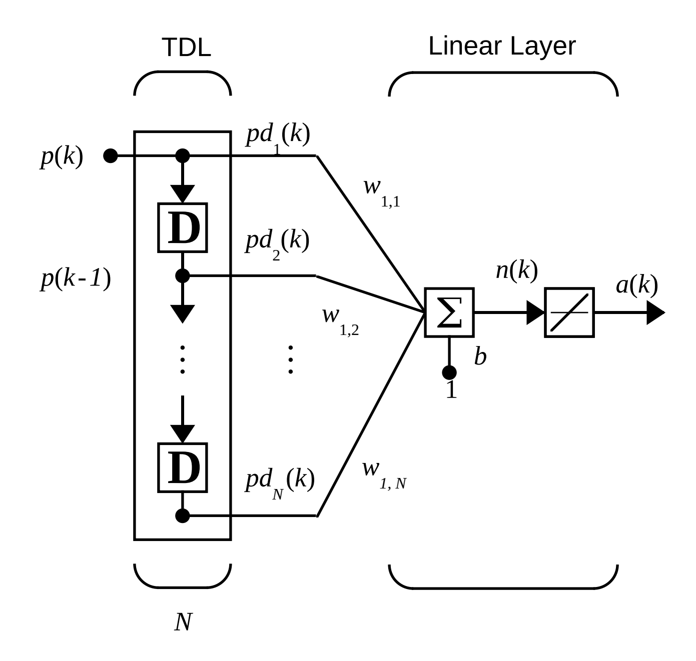
$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

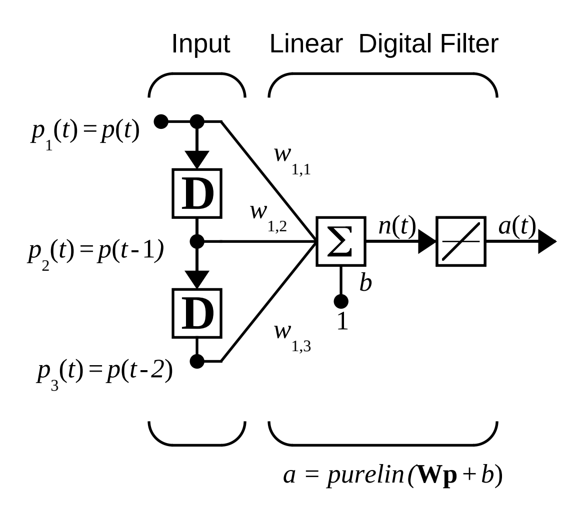
Tapped Delay Line



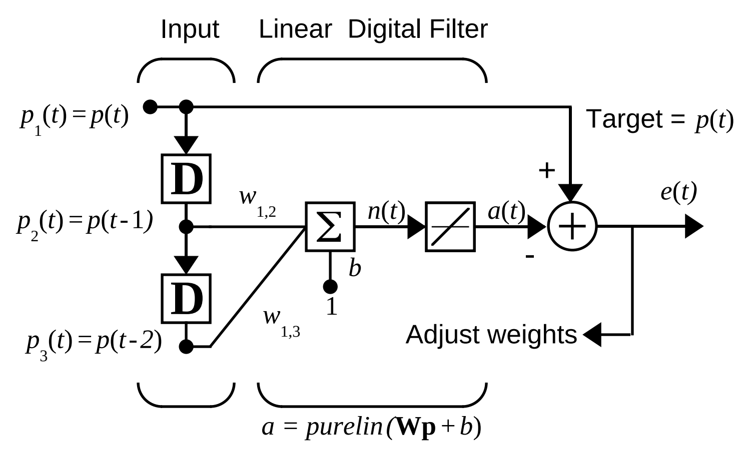
Adaptive Filter



Adaptive Filter Example

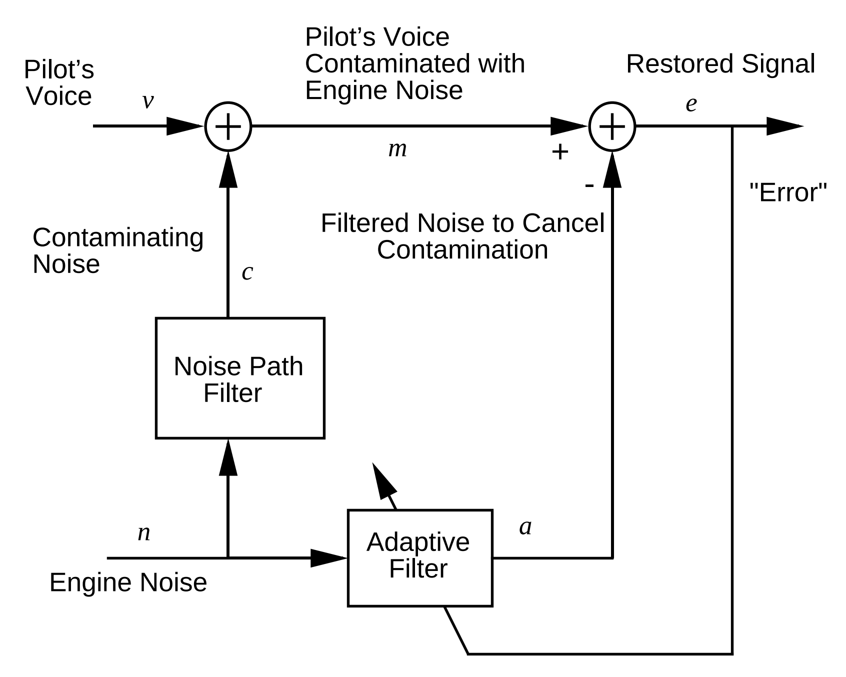


Prediction Example



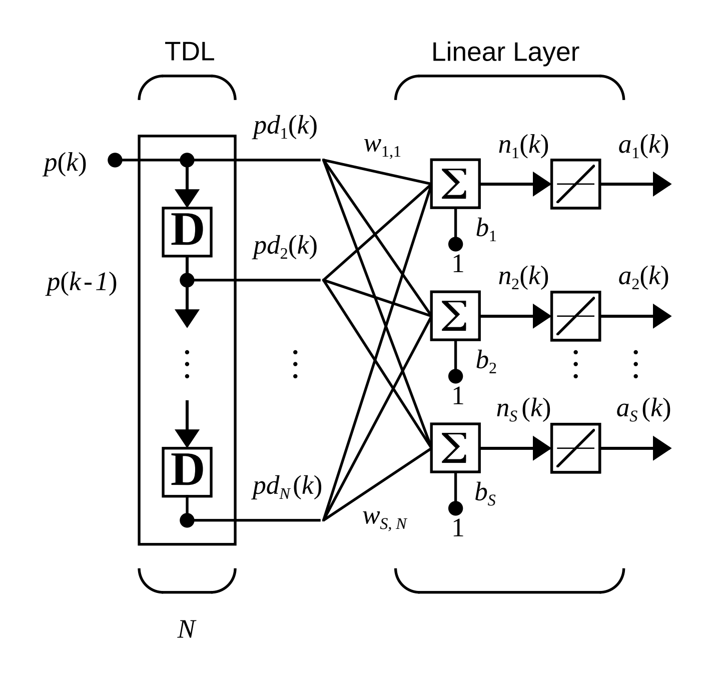
Predictive Filter: $a(t)$ is approximation to $p(t)$

Noise Cancellation Example

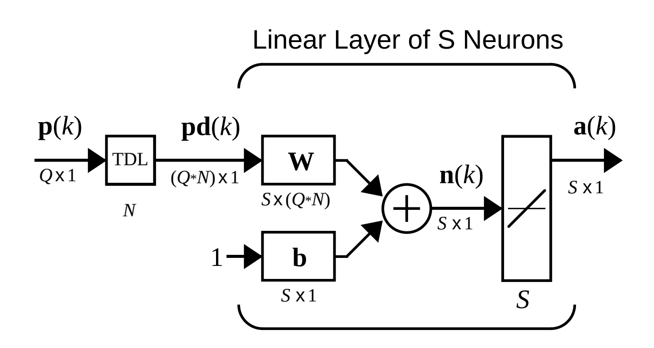


Adaptive Filter Adjusts to Minimize Error.
This removes the engine noise from contaminated signal, leaving the pilot's voice as the "error."

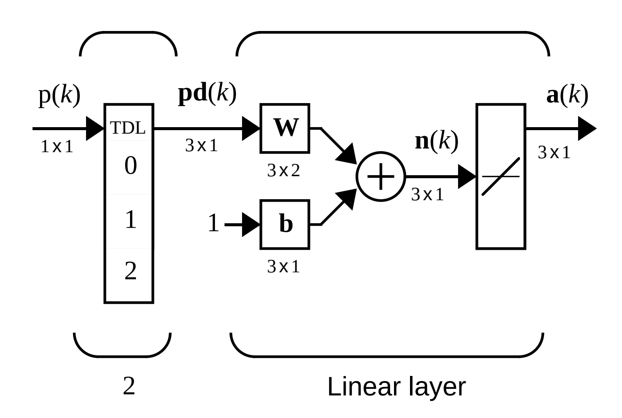
Multiple Neuron Adaptive Filter



Abbreviated Form of Adaptive Filter



Small Specific Adaptive Filter



New Functions

This chapter introduces the following new function.

Function	Description
`adapt`	Trains a network using a sequence of inputs

11

Applications

Introduction 11-2
Application Scripts 11-2

Applin1: Linear Design 11-3
Problem Definition 11-3
Network Design 11-4
Network Testing 11-4
Thoughts and Conclusions 11-6

Applin2: Adaptive Prediction 11-7
Problem Definition 11-7
Network Initialization 11-8
Network Training 11-8
Network Testing 11-8
Thoughts and Conclusions 11-10

Appelm1: Amplitude Detection 11-11
Problem Definition 11-11
Network Initialization 11-11
Network Training 11-12
Network Testing 11-13
Network Generalization 11-13
Improving Performance 11-15

Appcr1: Character Recognition 11-16
Problem Statement 11-16
Neural Network 11-17
System Performance 11-20
Summary 11-22

Introduction

Today, neural networks can solve problems of economic importance that could not be approached previously in any practical way. Some of the recent neural network applications are discussed in this chapter. See Chapter 1 for a list of many areas where neural networks already have been applied.

Note The rest of this chapter describes applications that are practical and make extensive use of the neural network functions described in this User's Guide.

Application Scripts

The linear network applications are contained in scripts `applin1` and `applin2`.

The Elman network amplitude detection application is contained in the script `appelm1`.

The character recognition application is in `appcr1`.

Type `help nndemos` to see a listing of all neural network demonstrations or applications.

Applin1: Linear Design

Problem Definition

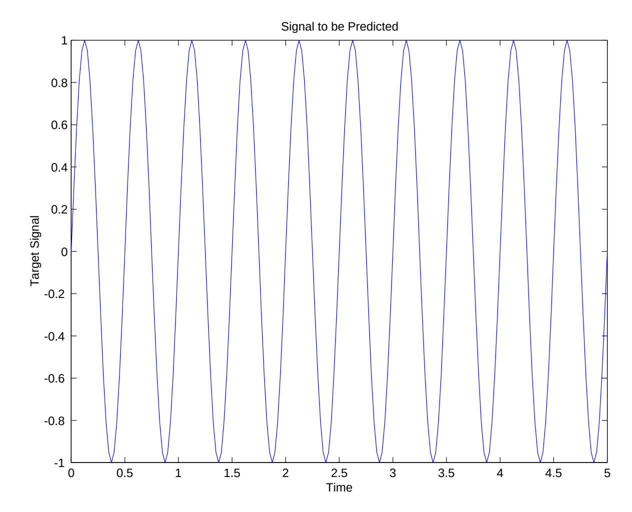
Here is the definition of a signal T, which lasts 5 seconds, and is defined at a sampling rate of 40 samples per second.

```
time = 0:0.025:5;
T = sin(time*4*pi);
Q = length(T);
```

At any given time step, the network is given the last five values of the signal t, and expected to give the next value. The inputs P are found by delaying the signal T from one to five time steps.

```
P = zeros(5,Q);
P(1,2:Q) = T(1,1:(Q-1));
P(2,3:Q) = T(1,1:(Q-2));
P(3,4:Q) = T(1,1:(Q-3));
P(4,5:Q) = T(1,1:(Q-4));
P(5,6:Q) = T(1,1:(Q-5));
```

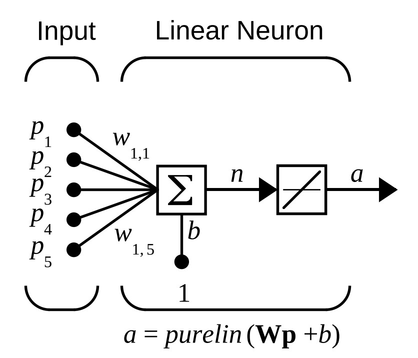
Here is a plot of the signal T.



Network Design

Because the relationship between past and future values of the signal is not changing, the network can be designed directly from examples using `newlind`.

The problem as defined above has five inputs (the five delayed signal values), and one output (the next signal value). Thus, the network solution must consist of a single neuron with five inputs.



Here `newlind` finds the weights and biases, for the neuron above, that minimize the sum-squared error for this problem.

```
net = newlind(P,T);
```

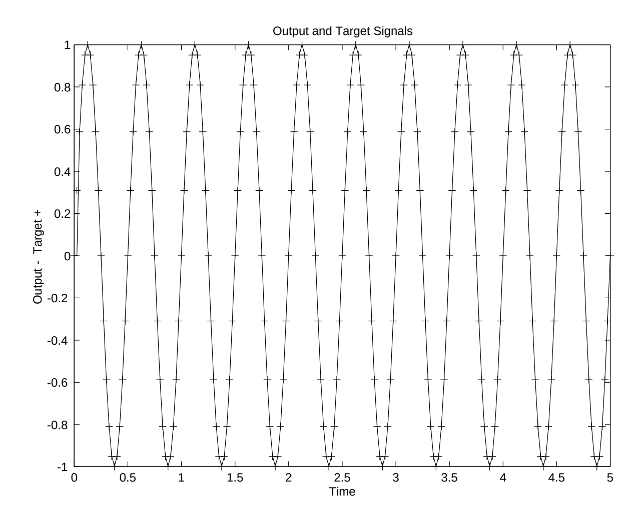
The resulting network can now be tested.

Network Testing

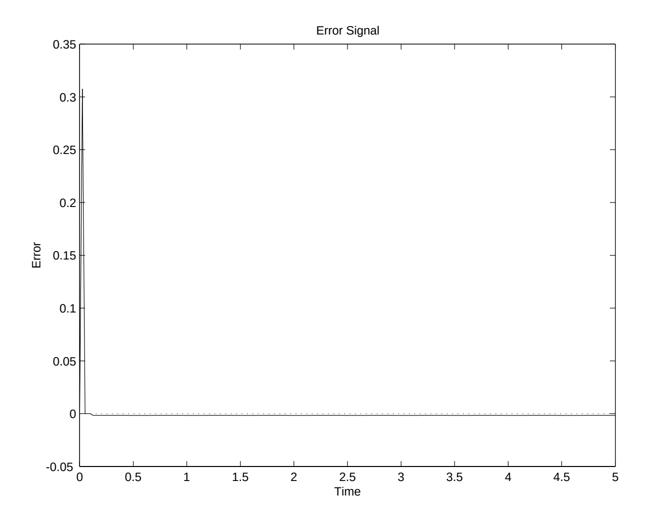
To test the network, its output `a` is computed for the five delayed signals `P` and compared with the actual signal `T`.

```
a = sim(net,P);
```

Here is a plot of `a` compared to `T`.



The network's output a and the actual signal t appear to match up perfectly. Just to be sure, let us plot the error e = T – a.



The network did have some error for the first few time steps. This occurred because the network did not actually have five delayed signal values available until the fifth time step. However, after the fifth time step error was negligible. The linear network did a good job. Run the script applin1 to see these plots.

Thoughts and Conclusions

While `newlind` is not able to return a zero error solution for nonlinear problems, it does minimize the sum-squared error. In many cases, the solution, while not perfect, may model a nonlinear relationship well enough to meet the application specifications. Giving the linear network many delayed signal values gives it more information with which to find the lowest error linear fit for a nonlinear problem.

Of course, if the problem is very nonlinear and/or the desired error is very low, backpropagation or radial basis networks would be more appropriate.

Applin2: Adaptive Prediction

In application script `applin2`, a linear network is trained incrementally with `adapt` to predict a time series. Because the linear network is trained incrementally, it can respond to changes in the relationship between past and future values of the signal.

Problem Definition

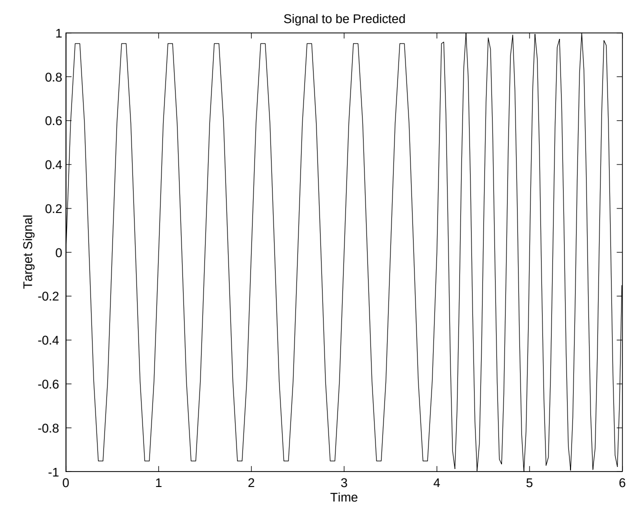
The signal `T` to be predicted lasts 6 seconds with a sampling rate of 20 samples per second. However, after 4 seconds the signal's frequency suddenly doubles.

```
time1 = 0:0.05:4;
time2 = 4.05:0.024:6;
time = [time1 time2];
T = [sin(time1*4*pi) sin(time2*8*pi)];
```

Since we are training the network incrementally, we change `t` to a sequence.

```
T = con2seq(T);
```

Here is a plot of this signal.

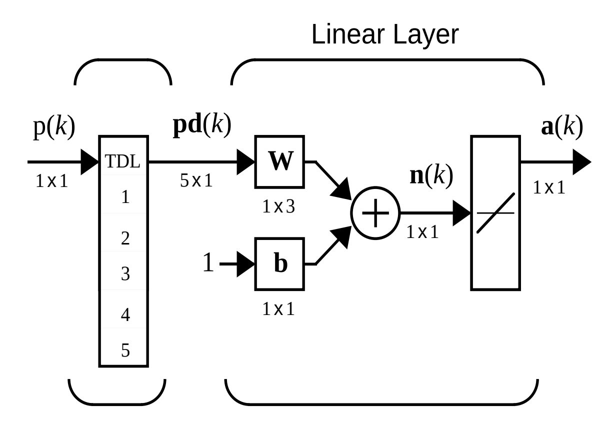


The input to the network is the same signal that makes up the target.

```
P = T;
```

Network Initialization

The network has only one neuron, as only one output value of the signal `T` is being generated at each time step. This neuron has five inputs, the five delayed values of the signal `T`.



The function `newlin` creates the network shown above. We use a learning rate of 0.1 for incremental training.

```
lr = 0.1;
delays = [1 2 3 4 5];
net = newlin(minmax(cat(2,P{:})),1,delays,lr);
[w,b] = initlin(P,t)
```

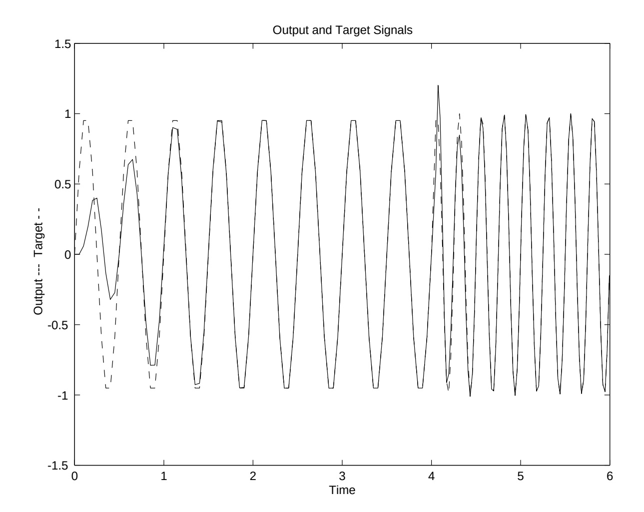
Network Training

The above neuron is trained incrementally with `adapt`. Here is the code to train the network on input/target signals `P` and `T`.

```
[net,a,e]=adapt(net,P,T);
```

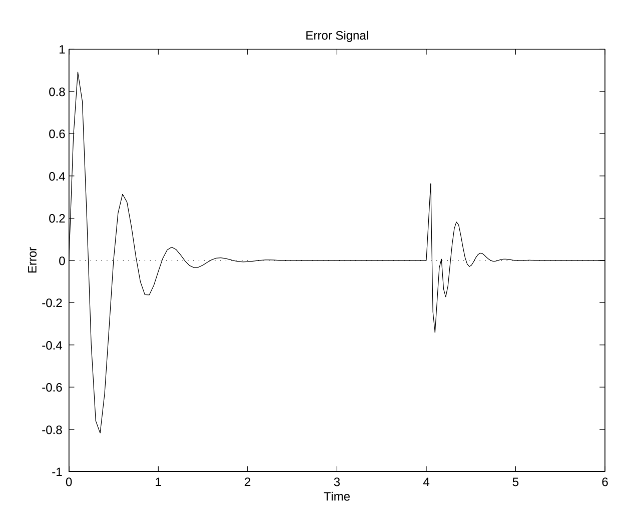
Network Testing

Once the network is adapted, we can plot its output signal and compare it to the target signal.



Initially, it takes the network 1.5 seconds (30 samples) to track the target signal. Then, the predictions are accurate until the fourth second when the target signal suddenly changes frequency. However, the adaptive network learns to track the new signal in an even shorter interval as it has already learned a behavior (a sine wave) similar to the new signal.

A plot of the error signal makes these effects easier to see.



Thoughts and Conclusions

The linear network was able to adapt very quickly to the change in the target signal. The 30 samples required to learn the wave form are very impressive when one considers that in a typical signal processing application, a signal may be sampled at 20 kHz. At such a sampling frequency, 30 samples go by in 1.5 milliseconds.

For example, the adaptive network can be monitored so as to give a warning that its constants were nearing values that would result in instability.

Another use for an adaptive linear model is suggested by its ability to find a minimum sum-squared error linear estimate of a nonlinear system's behavior. An adaptive linear model is highly accurate as long as the nonlinear system stays near a given operating point. If the nonlinear system moves to a different operating point, the adaptive linear network changes to model it at the new point.

The sampling rate should be high to obtain the linear model of the nonlinear system at its current operating point in the shortest amount of time. However, there is a minimum amount of time that must occur for the network to see enough of the system's behavior to properly model it. To minimize this time, a small amount of noise can be added to the input signals of the nonlinear system. This allows the network to adapt faster as more of the operating points dynamics are expressed in a shorter amount of time. Of course, this noise should be small enough so it does not affect the system's usefulness.

Appelm1: Amplitude Detection

Elman networks can be trained to recognize and produce both spatial and temporal patterns. An example of a problem where temporal patterns are recognized and classified with a spatial pattern is amplitude detection.

Amplitude detection requires that a wave form be presented to a network through time, and that the network output the amplitude of the wave form. This is not a difficult problem, but it demonstrates the Elman network design process.

The following material describes code that is contained in the demonstration script `appelm1`.

Problem Definition

The following code defines two sine wave forms, one with an amplitude of 1.0, the other with an amplitude of 2.0.

```
p1 = sin(1:20);
p2 = sin(1:20)*2;
```

The target outputs for these wave forms is their amplitudes.

```
t1 = ones(1,20);
t2 = ones(1,20)*2;
```

These wave forms can be combined into a sequence where each wave form occurs twice. These longer wave forms are used to train the Elman network.

```
p = [p1 p2 p1 p2];
t = [t1 t2 t1 t2];
```

We want the inputs and targets to be considered a sequence, so we need to make the conversion from the matrix format.

```
Pseq = con2seq(p);
Tseq = con2seq(t);
```

Network Initialization

This problem requires that the Elman network detect a single value (the signal), and output a single value (the amplitude), at each time step. Therefore the network must have one input element, and one output neuron.

```
R = 1;% 1 input element
S2 = 1;% 1 layer 2 output neuron
```

The recurrent layer can have any number of neurons. However, as the complexity of the problem grows, more neurons are needed in the recurrent layer for the network to do a good job.

This problem is fairly simple, so only 10 recurrent neurons are used in the first layer.

```
S1 = 10;% 10 recurrent neurons in the first layer
```

Now the function `newelm` can be used to create initial weight matrices and bias vectors for a network with one input that can vary between –2 and +2. We use variable learning rate (`traingdx`) for this example.

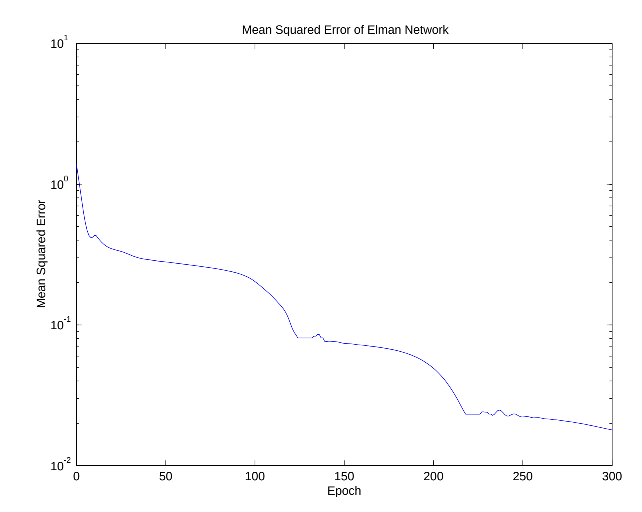
```
net = newelm([-2 2],[S1 S2],{'tansig','purelin'},'traingdx');
```

Network Training

Now call `train`.

```
[net,tr] = train(net,Pseq,Tseq);
```

As this function finishes training at 500 epochs, it displays the following plot of errors.



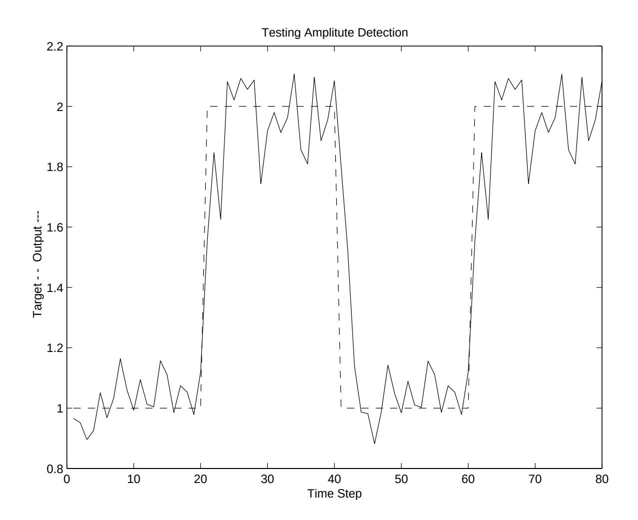
The final mean-squared error was about 1.8e-2. We can test the network to see what this means.

Network Testing

To test the network, the original inputs are presented, and its outputs are calculated with `simuelm`.

```
a = sim(net,Pseq);
```

Here is the plot.



The network does a good job. New wave amplitudes are detected with a few samples. More neurons in the recurrent layer and longer training times would result in even better performance.

The network has successfully learned to detect the amplitudes of incoming sine waves.

Network Generalization

Of course, even if the network detects the amplitudes of the training wave forms, it may not detect the amplitude of a sine wave with an amplitude it has not seen before.

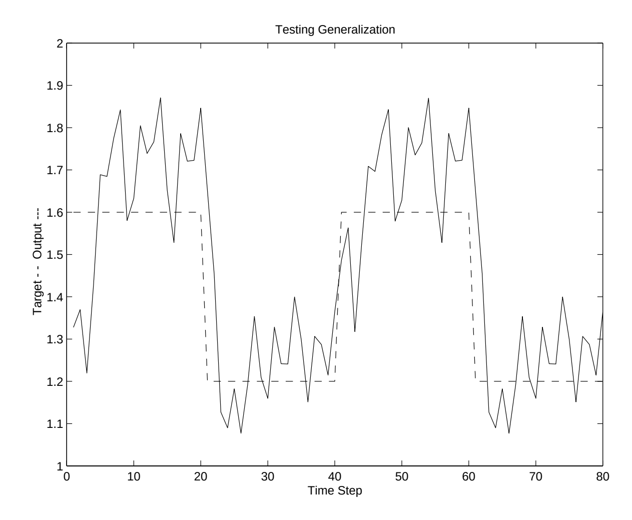
The following code defines a new wave form made up of two repetitions of a sine wave with amplitude 1.6 and another with amplitude 1.2.

```
p3 = sin(1:20)*1.6;
t3 = ones(1,20)*1.6;
p4 = sin(1:20)*1.2;
t4 = ones(1,20)*1.2;
pg = [p3 p4 p3 p4];
tg = [t3 t4 t3 t4];
pgseq = con2seq(pg);
```

The input sequence pg and target sequence tg are used to test the ability of our network to generalize to new amplitudes.

Once again the function sim is used to simulate the Elman network and the results are plotted.

```
a = sim(net,pgseq);
```



This time the network did not do as well. It seems to have a vague idea as to what it should do, but is not very accurate!

Improved generalization could be obtained by training the network on more amplitudes than just 1.0 and 2.0. The use of three or four different wave forms with different amplitudes can result in a much better amplitude detector.

Improving Performance

Run `appelm1` to see plots similar to those above. Then make a copy of this file and try improving the network by adding more neurons to the recurrent layer, using longer training times, and giving the network more examples in its training data.

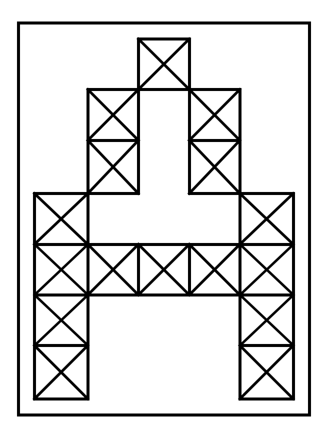
Appcr1: Character Recognition

It is often useful to have a machine perform pattern recognition. In particular, machines that can read symbols are very cost effective. A machine that reads banking checks can process many more checks than a human being in the same time. This kind of application saves time and money, and eliminates the requirement that a human perform such a repetitive task. The script `appcr1` demonstrates how character recognition can be done with a backpropagation network.

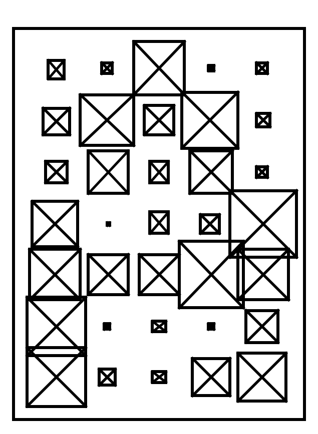
Problem Statement

A network is to be designed and trained to recognize the 26 letters of the alphabet. An imaging system that digitizes each letter centered in the system's field of vision is available. The result is that each letter is represented as a 5 by 7 grid of boolean values.

For example, here is the letter A.



However, the imaging system is not perfect and the letters may suffer from noise.



Perfect classification of ideal input vectors is required, and reasonably accurate classification of noisy vectors.

The twenty-six 35-element input vectors are defined in the function `prprob` as a matrix of input vectors called `alphabet`. The target vectors are also defined in this file with a variable called `targets`. Each target vector is a 26-element vector with a 1 in the position of the letter it represents, and 0's everywhere else. For example, the letter A is to be represented by a 1 in the first element (as A is the first letter of the alphabet), and 0's in elements two through twenty-six.

Neural Network

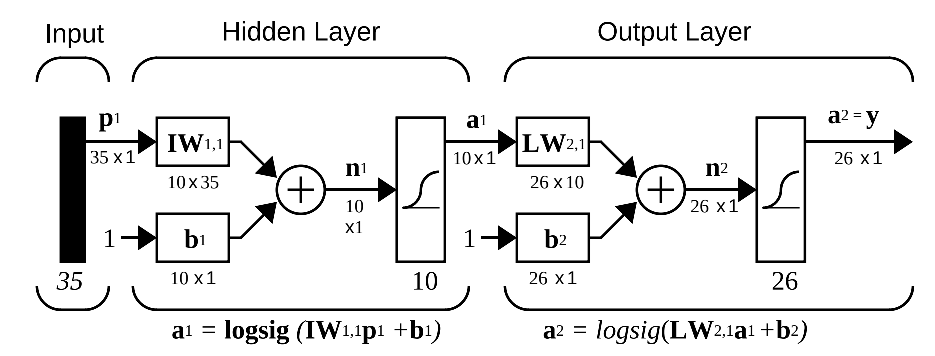
The network receives the 35 Boolean values as a 35-element input vector. It is then required to identify the letter by responding with a 26-element output vector. The 26 elements of the output vector each represent a letter. To operate correctly, the network should respond with a 1 in the position of the letter being presented to the network. All other values in the output vector should be 0.

In addition, the network should be able to handle noise. In practice, the network does not receive a perfect Boolean vector as input. Specifically, the network should make as few mistakes as possible when classifying vectors with noise of mean 0 and standard deviation of 0.2 or less.

Architecture

The neural network needs 35 inputs and 26 neurons in its output layer to identify the letters. The network is a two-layer `log-sigmoid/log-sigmoid`

network. The log-sigmoid transfer function was picked because its output range (0 to 1) is perfect for learning to output boolean values.



The hidden (first) layer has 10 neurons. This number was picked by guesswork and experience. If the network has trouble learning, then neurons can be added to this layer.

The network is trained to output a 1 in the correct position of the output vector and to fill the rest of the output vector with 0's. However, noisy input vectors may result in the network not creating perfect 1's and 0's. After the network is trained the output is passed through the competitive transfer function `compet`. This makes sure that the output corresponding to the letter most like the noisy input vector takes on a value of 1, and all others have a value of 0. The result of this post-processing is the output that is actually used.

Initialization

The two-layer network is created with `newff`.

```
S1 = 10;
[R,Q] = size(alphabet);
[S2,Q] = size(targets);
P = alphabet;
net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');
```

Training

To create a network that can handle noisy input vectors it is best to train the network on both ideal and noisy vectors. To do this, the network is first trained on ideal vectors until it has a low sum-squared error.

Then, the network is trained on 10 sets of ideal and noisy vectors. The network is trained on two copies of the noise-free alphabet at the same time as it is trained on noisy vectors. The two copies of the noise-free alphabet are used to maintain the network's ability to classify ideal input vectors.

Unfortunately, after the training described above the network may have learned to classify some difficult noisy vectors at the expense of properly classifying a noise-free vector. Therefore, the network is again trained on just ideal vectors. This ensures that the network responds perfectly when presented with an ideal letter.

All training is done using backpropagation with both adaptive learning rate and momentum with the function `trainbpx`.

Training Without Noise

The network is initially trained without noise for a maximum of 5000 epochs or until the network sum-squared error falls beneath 0.1.

```
P = alphabet;
T = targets;
net.performFcn = 'sse';
net.trainParam.goal = 0.1;
net.trainParam.show = 20;
net.trainParam.epochs = 5000;
net.trainParam.mc = 0.95;
[net,tr] = train(net,P,T);
```

Training with Noise

To obtain a network not sensitive to noise, we trained with two ideal copies and two noisy copies of the vectors in `alphabet`. The target vectors consist of four copies of the vectors in `target`. The noisy vectors have noise of mean 0.1 and 0.2 added to them. This forces the neuron to learn how to properly identify noisy letters, while requiring that it can still respond well to ideal vectors.

To train with noise, the maximum number of epochs is reduced to 300 and the error goal is increased to 0.6, reflecting that higher error is expected because more vectors (including some with noise), are being presented.

```
netn = net;
netn.trainParam.goal = 0.6;
netn.trainParam.epochs = 300;
```

```
T = [targets targets targets targets];
for pass = 1:10
P = [alphabet, alphabet, ...
      (alphabet + randn(R,Q)*0.1), ...
      (alphabet + randn(R,Q)*0.2)];
[netn,tr] = train(netn,P,T);
end
```

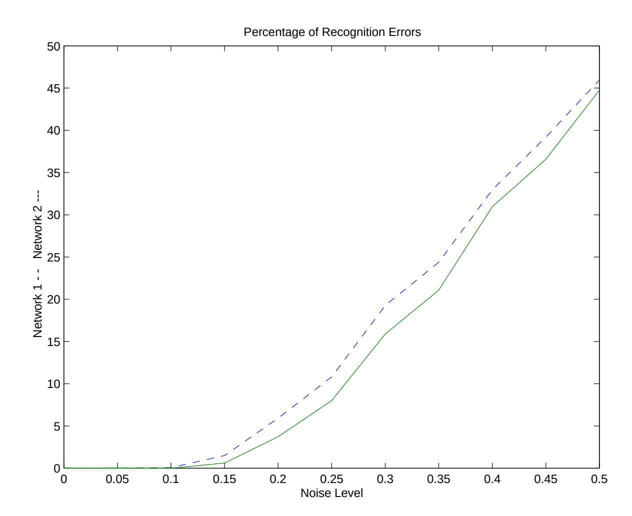
Training Without Noise Again

Once the network is trained with noise, it makes sense to train it without noise once more to ensure that ideal input vectors are always classified correctly. Therefore, the network is again trained with code identical to the "Training Without Noise" section.

System Performance

The reliability of the neural network pattern recognition system is measured by testing the network with hundreds of input vectors with varying quantities of noise. The script file appcr1 tests the network at various noise levels, and then graphs the percentage of network errors versus noise. Noise with a mean of 0 and a standard deviation from 0 to 0.5 is added to input vectors. At each noise level, 100 presentations of different noisy versions of each letter are made and the network's output is calculated. The output is then passed through the competitive transfer function so that only one of the 26 outputs (representing the letters of the alphabet), has a value of 1.

The number of erroneous classifications is then added and percentages are obtained.



The solid line on the graph shows the reliability for the network trained with and without noise. The reliability of the same network when it had only been trained without noise is shown with a dashed line. Thus, training the network on noisy input vectors greatly reduces its errors when it has to classify noisy vectors.

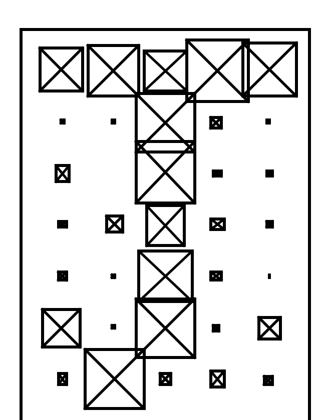
The network did not make any errors for vectors with noise of mean 0.00 or 0.05. When noise of mean 0.2 was added to the vectors both networks began making errors.

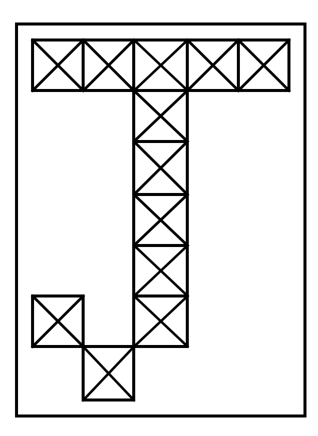
If a higher accuracy is needed, the network can be trained for a longer time or retrained with more neurons in its hidden layer. Also, the resolution of the input vectors can be increased to a 10-by-14 grid. Finally, the network could be trained on input vectors with greater amounts of noise if greater reliability were needed for higher levels of noise.

To test the system, a letter with noise can be created and presented to the network.

```
noisyJ = alphabet(:,10)+randn(35,1) * 0.2;
plotchar(noisyJ);
A2 = sim(net,noisyJ);
A2 = compet(A2);
answer = find(compet(A2) == 1);
plotchar(alphabet(:,answer));
```

Here is the noisy letter and the letter the network picked (correctly).





Summary

This problem demonstrates how a simple pattern recognition system can be designed. Note that the training process did not consist of a single call to a training function. Instead, the network was trained several times on various input vectors.

In this case, training a network on different sets of noisy vectors forced the network to learn how to deal with noise, a common problem in the real world.

12

Advanced Topics

Custom Networks 12-2
Custom Network 12-3
Network Definition 12-4
Network Behavior 12-12

Additional Toolbox Functions 12-16
Initialization Functions 12-16
Transfer Functions 12-16
Learning Functions 12-17

Custom Functions 12-18
Simulation Functions 12-18
Initialization Functions 12-24
Learning Functions 12-27
Self-Organizing Map Functions 12-36

Custom Networks

The Neural Network Toolbox is designed to allow for many kinds of networks. This makes it possible for many functions to use the same network object data type.

Here are all the standard toolbox network creation functions.

New Networks	
newc	Create a competitive layer.
newcf	Create a cascade-forward backpropagation network.
newelm	Create an Elman backpropagation network.
newff	Create a feed-forward backpropagation network.
newfftd	Create a feed-forward input-delay backprop network.
newgrnn	Design a generalized regression neural network.
newhop	Create a Hopfield recurrent network.
newlin	Create a linear layer.
newlind	Design a linear layer.
newlvq	Create a learning vector quantization network
newp	Create a perceptron.
newpnn	Design a probabilistic neural network.
newrb	Design a radial basis network.
newrbe	Design an exact radial basis network.
newsom	Create a self-organizing map.

This flexibility is possible because we have an object-oriented representation for networks. The representation allows various architectures to be defined and allows various algorithms to be assigned to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

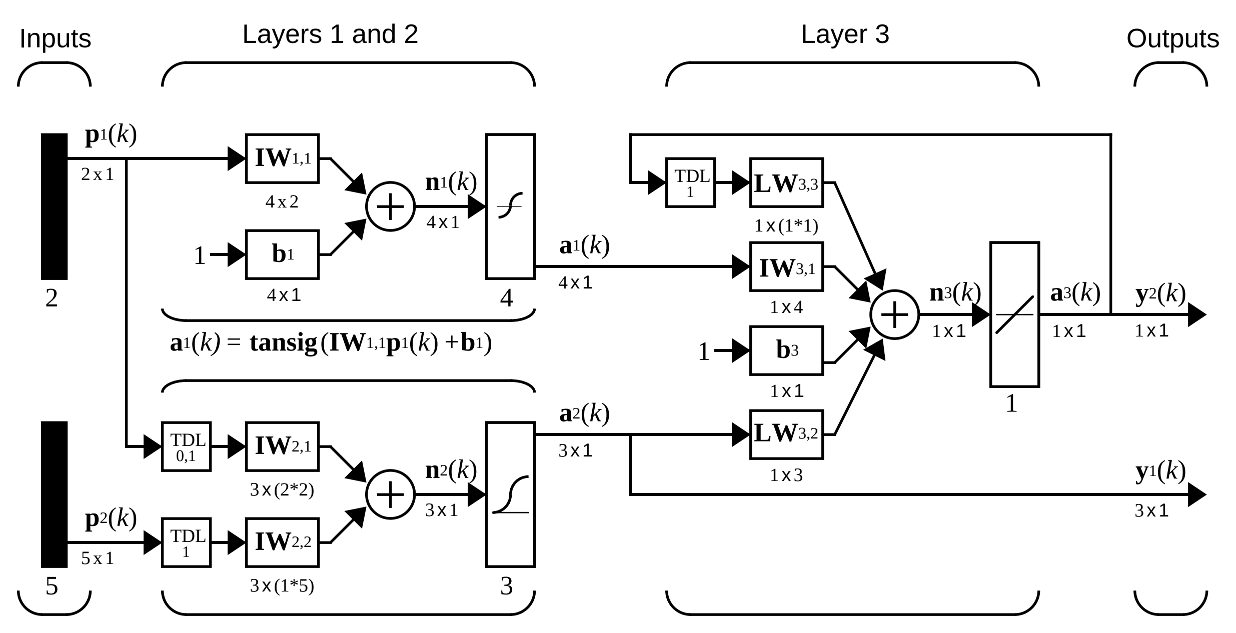
```
network  - Create a custom neural network.
```

The network object consists of many properties that you can set to specify the structure and behavior of your network. See Chapter 13, "" for descriptions of all network properties.

The following sections demonstrate how to create a custom network by using these properties.

Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input ranges from -2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

We agree here that each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method (`initnw`). Also, the network is trained with the Levenberg-Marquardt backpropagation (`trainlm`), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error (`mse`).

Network Definition

The first step is to create a new network. Type in the following code to create a network and view its many properties.

```
net = network
```

Architecture Properties

The first group of properties displayed are labeled `architecture` properties. These properties allow you to select of the number of inputs and layers, and their connections.

Number of Inputs and Layers. The first two properties displayed are `numInputs` and `numLayers`. These properties allow us to select how many inputs and layers we want our network to have.

```
net =

    Neural Network object:

    architecture:

          numInputs: 0
          numLayers: 0
          …
```

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in our custom network diagram.

```
net.numInputs = 2;
net.numLayers = 3;
```

Note that `net.numInputs` is the number of input sources, not the number of elements in an input vector (`net.inputs{i}.size`).

Bias Connections. Type `net` and press **Return** to view its properties again. The network now has two inputs and three layers.

```
net =

    Neural Network object:

    architecture:

         numInputs: 2
         numLayers: 3
```

Now look at the next five properties.

```
       biasConnect: [0; 0; 0]
      inputConnect: [0 0; 0 0; 0 0]
      layerConnect: [0 0 0; 0 0 0; 0 0 0]
     outputConnect: [0 0 0]
     targetConnect: [0 0 0]
```

These matrices of 1's and 0's represent the presence or absence of bias, input weight, layer weight, output, and target connections. They are currently all zeros, indicating that the network does not have any such connections.

Note that the bias connection matrix is a 3-by-1 vector. To create a bias connection to the *i*th layer you can set `net.biasConnect(i)` to `1`. Specify that the first and third layer's are to have bias connections, as our diagram indicates, by typing in the following code.

```
net.biasConnect(1) = 1;
net.biasConnect(3) = 1;
```

Note that you could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

Input and Layer Weight Connections. The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the *i*th layer from the *j*th input.

To connect the first input to the first and second layers, and the second input to the second layer (as is indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;
net.inputConnect(2,1) = 1;
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i.j)` represents the presence of a layer-weight connection going to the *i*th layer from the *j*th layer. Connect layers 1, 2, and 3 to layer 3 as follows.

```
net.layerConnect = [0 0 0; 0 0 0; 1 1 1];
```

Output and Target Connections. Both the output and target connection matrices are 1-by-3 matrices, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to network outputs, type

```
net.outputConnect = [0 1 1];
```

To give layer 3 a target connection, type

```
net.targetConnect = [0 0 1];
```

The layer 3 target is compared to the output of layer 3 to generate an error for use when measuring the performance of the network, or when updating the network during training or adaption.

Number of Outputs and Targets

Type `net` and press **Enter** to view the updated properties. The final four architecture properties are read-only values, which means their values are determined by the choices we make for other properties. The first two read-only properties have the following values.

```
numOutputs: 2  (read-only)
numTargets: 1  (read-only)
```

By defining output connections from layers 2 and 3, and a target connection from layer 3, you specify that the network has two outputs and one target.

Subobject Properties

The next group of properties is

```
subobject structures:

            inputs: {2x1 cell} of inputs
            layers: {3x1 cell} of layers
           outputs: {1x3 cell} containing 2 outputs
           targets: {1x3 cell} containing 1 target
            biases: {3x1 cell} containing 2 biases
      inputWeights: {3x2 cell} containing 3 input weights
      layerWeights: {3x3 cell} containing 3 layer weights
```

Inputs

When you set the number of inputs (`net.numInputs`) to 2, the `inputs` property becomes a cell array of two input structures. Each *i*th input structure (`net.inputs{i}`) contains addition properties associated with the *i*th input.

To see how the input structures are arranged, type

```
net.inputs
ans =

    [1x1 struct]
    [1x1 struct]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows.

```
ans =

       range: [0 1]
        size: 1
    userdata: [1x1 struct]
```

Note that the `range` property only has one row. This indicates that the input has only one element, which varies from 0 to 1. The `size` property also indicates that this input has just one element.

The first input vector of the custom network is to have two elements ranging from 0 to 10. Specify this by altering the range property of the first input as follows.

```
net.inputs{1}.range = [0 10; 0 10];
```

If we examine the first input's structure again, we see that it now has the correct size, which was inferred from the new `range` values.

```
ans =

       range: [2x2 double]
        size: 2
    userdata: [1x1 struct]
```

Set the second input vector ranges to be from -2 to 2 for five elements as follows.

```
net.inputs{2}.range = [-2 2; -2 2; -2 2; -2 2; -2 2];
```

Layers. When we set the number of layers (`net.numLayers`) to 3, the layers property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}

ans =

     dimensions: 1
    distanceFcn: 'dist'
      distances: 0
        initFcn: 'initwb'
    netInputFcn: 'netsum'
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'purelin'
       userdata: [1x1 struct]
```

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to `tansig`, and its initialization function to the Nguyen-Widrow function as required for the custom network diagram.

```
net.layers{1}.size = 4;
net.layers{1}.transferFcn =  tansig ;
```

```
net.layers{1}.initFcn =  initnw ;
```

The second layer is to have three neurons, the `logsig` transfer function, and be initialized with `initnw`. Thus, set the second layer's properties to the desired values as follows.

```
net.layers{2}.size = 3;
net.layers{2}.transferFcn =  logsig ;
net.layers{2}.initFcn =  initnw ;
```

The third layer's size and transfer function properties don't need to be changed since the defaults match those shown in the network diagram. You only need to set its initialization function as follows.

```
net.layers{3}.initFcn =  initnw ;
```

Output and Targets. Take a look at how the `outputs` property is arranged with this line of code.

```
net.outputs
ans =

     []    [1x1 struct]    [1x1 struct]
```

Note that `outputs` contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when `net.outputConnect` was set to `[0 1 1]`.

View the second layer's output structure with the following expression.

```
net.outputs{2}
ans =

        size: 3
    userdata: [1x1 struct]
```

The `size` is automatically set to 3 when the second layer's size (`net.layers{2}.size`) is set to that value. Take a look at the third layer's output structure if you want to verify that it also has the correct `size`.

Similarly, `targets` contains one structure representing the third layer's target. Type these two lines of code to see how `targets` is arranged and to view the third layer's target properties.

```
net.targets
```

```
ans =
     []    []    [1x1 struct]
```

```
net.targets{3}
ans =
        size: 1
    userdata: [1x1 struct]
```

Biases, Input Weights, and Layer Weights. Enter the following lines of code to see how bias and weight structures are arranged.

```
net.biases
net.inputWeights
net.layerWeights
```

Here are the results for typing `net.biases`.

```
ans =
    [1x1 struct]
              []
    [1x1 struct]
```

If you examine the results you will note that each contains a structure where the corresponding connections (`net.biasConnect`, `net.inputConnect`, and `net.layerConnect`) contain a 1.

Take a look at their structures with these lines of code.

```
net.biases{1}
net.biases{3}
net.inputWeights{1,1}
net.inputWeights{2,1}
net.inputWeights{2,2}
net.layerWeights{3,1}
net.layerWeights{3,2}
net.layerWeights{3,3}
```

For example, typing `net.biases{1}` results in the following output.

```
ans =
       initFcn: ''
         learn: 1
      learnFcn: ''
    learnParam: ''
```

```
            size: 4
        userdata: [1x1 struct]
```

Specify the weights tap delay lines in accordance with the network diagram, by setting each weights `delays` property.

```
net.inputWeights{2,1}.delays = [0 1];
net.inputWeights{2,2}.delays = 1;
net.layerWeights{3,3}.delays = 1;
```

Network Functions

Type `net` and press return again to see the next set of properties.

```
    functions:

            adaptFcn: (none)
             initFcn: (none)
          performFcn: (none)
            trainFcn: (none)
```

Each of these properties defines a function for a basic network operation.

Set the initialization function to `initlay` so the network initializes itself according to the layer initialization functions that we have already set to `initnw` the Nguyen-Widrow initialization function.

```
net.initFcn = 'initlay';
```

This meets the initialization requirement of our network.

Set the performance function to `mse` (mean squared error) and the training function to `trainlm` (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = 'mse';
net.trainFcn = 'trainlm';
```

Weight and Bias Values

Before initializing and training the network, take a look at the final group of network properties (aside from the `userdata` property).

```
weight and bias values:

                  IW: {3x2 cell} containing 3 input weight matrices
```

```
            LW: {3x3 cell} containing 3 layer weight matrices
             b: {3x1 cell} containing 2 bias vectors
```

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (`net.inputConnect`, `net.layerConnect`, `net.biasConnect`) contain 1's and the subobject properties (`net.inputWeights`, `net.layerWeights`, `net.biases`) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.IW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

Each input weight `net.IW{i,j}`, layer weight `net.LW{i,j}`, and bias vector `net.b{i}` has as many rows as the size of the *i*th layer (`net.layers{i}.size`).

Each input weight `net.IW{i,j}` has as many columns as the size of the *j*th input (`net.inputs{j}.size`) multiplied by the number of its delay values (`length(net.inputWeights{i,j}.delays)`).

Likewise, each layer weight has as many columns as the size of the *j*th layer (`net.layers{j}.size`) multiplied by the number of its delay values (`length(net.layerWeights{i,j}.delays)`).

Network Behavior

Initialization

Initialize your network with the following line of code.

```
net = init(net)
```

Reference the network's biases and weights again to see how they have changed.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.IW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

For example,

```
net.IW{1,1}
```

```
ans =
   -0.3040    0.4703
   -0.5423   -0.1395
    0.5567    0.0604
    0.2667    0.4924
```

Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

```
P = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1] [-1; -1; 1; 0; 1]}
```

We want the network to respond with the following target sequence.

```
T = {1 -1}
```

Before training, we can simulate the network to see whether the initial network's response Y is close to the target T.

```
Y = sim(net,P)

Y =

    [3x1 double]    [3x1 double]
    [    0.0456]    [    0.2119]
```

The second row of the cell array `Y` is the output sequence of the second network output, which is also the output sequence of the third layer. The values you got for the second row may differ from those shown due to different initial weights and biases. However, they will almost certainly not be equal to our targets T, which is also true of the values shown.

The next task is to prepare the training parameters. The following line of code displays the default Levenberg-Marquardt training parameters (which were defined when we set `net.trainFcn` to `trainlm`).

```
net.trainParam
```

The following properties should be displayed.

```
ans =

       epochs: 100
         goal: 0
     max_fail: 5
```

```
mem_reduc: 1
 min_grad: 1.0000e-10
       mu: 1.0000e-03
   mu_dec: 0.1000
   mu_inc: 10
   mu_max: 1.0000e+10
     show: 25
     time:
```

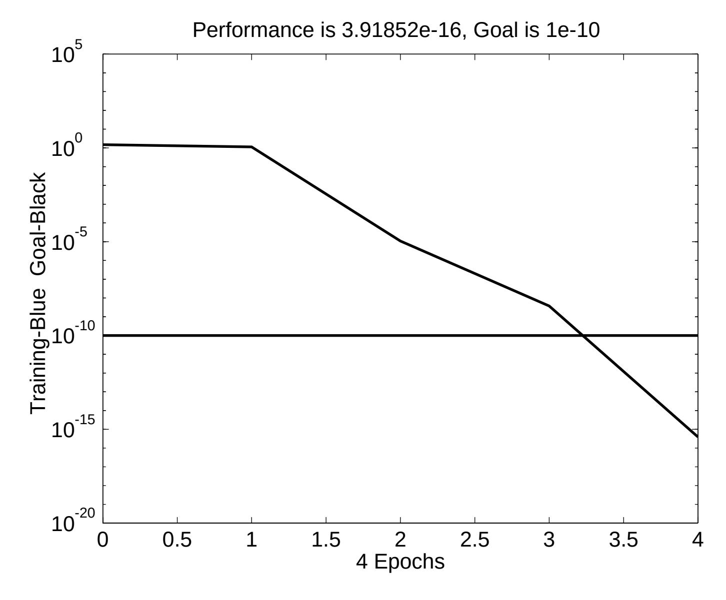
Change the performance goal to `1e-10`.

```
net.trainParam.goal = 1e-10;
```

Next, train the network with the following call.

```
net = train(net,P,T);
```

Below is a typical training plot.



After training you can simulate the network to see if it has learned to respond correctly.

```
Y = sim(net,P)
```

```
Y =

    [3x1 double]    [3x1 double]
    [    1.0000]    [   -1.0000]
```

Note that the second network output (i.e., the second row of the cell array Y), which is also the third layer's output, does match the target sequence T.

Additional Toolbox Functions

Most toolbox functions are explained in chapters dealing with networks that use them. However, some functions are not used by toolbox networks, but are included as they may be useful to you in creating custom networks.

Each of these is documented in “Reference” in Chapter 14. However, the notes given below may also prove to be helpful.

Initialization Functions

randnc

This weight initialization function generates random weight matrices whose columns are normalized to a length of 1.

randnr

This weight initialization function generates random weight matrices whose rows are normalized to a length of 1.

Transfer Functions

satlin

This transfer function is similar to `satlins`, but has a linear region going from 0 to 1 (instead of -1 to 1), and minimum and maximum values of 0 and 1 (instead of -1 and 1).

softmax

This transfer function is a softer version of the hard competitive transfer function `compet`. The neuron with the largest net input gets an output closest to one, while other neurons have outputs close to zero.

tribas

The triangular-basis transfer function is similar to the radial-basis transfer function `radbas`, but has a simpler shape.

Learning Functions

learnh

The Hebb weight learning function increases weights in proportion to the product, the weights input, and the neuron's output. This allows neurons to learn associations between their inputs and outputs.

learnhd

The Hebb-with-decay learning function is similar to the Hebb function, but adds a term that decreases weights each time step exponentially. This weight decay allows neurons to forget associations that are not reinforced regularly, and solves the problem that the Hebb function has with weights growing without bounds.

learnis

The instar weight learning function moves a neuron's weight vector towards the neuron's input vector with steps proportional to the neuron's output. This function allows neurons to learn association between input vectors and their outputs.

learnos

The outstar weight learning function acts in the opposite way as the instar learning rule. The outstar rule moves the weight vector coming from an input toward the output vector of a layer of neurons with step sizes proportional to the input value. This allows inputs to learn to recall vectors when stimulated.

Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) `A`, given its net input vector (or matrix) `N`. The only constraint on the relationship between the

output and net input is that the output must have the same dimensions as the input.

Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function `yourtf` to the second layer of a network.

```
net.layers{2}.transferFcn = 'yourtf';
```

Your transfer function then is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid transfer function, your function must calculate outputs `A` from net inputs `N` as follows,

```
A = yourtf(N)
```

where:

- `N` is an S x Q matrix of Q net input (column) vectors.
- `A` is an S x Q matrix of Q output (column) vectors.

Your transfer function must also provide information about itself, using this calling format,

```
info = yourtf(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'deriv'` - Returns the name of the associated derivative function.
- `'output'` - Returns the output range.
- `'active'` - Returns the active input range.

The toolbox contains an example custom transfer function called `mytf`. Enter the following lines of code to see how it is used.

```
help mytf
n = -5:.1:5;
a = mytf(n);
plot(n,a)
mytf('deriv')
```

Enter the following command to see how mytf is implemented.

```
type mytf
```

You can use mytf as a template to create your own transfer function.

Transfer Derivative Functions. If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,

```
dA_dN = yourdtf(N,A)
```

where:

- N is an $S \times Q$ matrix of Q net input (column) vectors.
- A is an $S \times Q$ matrix of Q output (column) vectors.
- dA_dN is the $S \times Q$ derivative dA/dN.

This only works for transfer functions whose output elements are independent. In other words, where each A(i) is only a function of N(i). Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.

To see how the example custom transfer derivative function mydtf works, type

```
help mydtf
da_dn = mydtf(n,a)
subplot(2,1,1), plot(n,a)
subplot(2,1,2), plot(n,dn_da)
```

Use this command to see how mydtf was implemented.

```
type mydtf
```

You can use mydtf as a template to create your own transfer derivative functions.

Net Input Functions

Net input functions calculate a layer's net input vector (or matrix) N, given its weighted input vectors (or matrices) Zi. The only constraints on the relationship between the net input and the weighted inputs are that the net

input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.

Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.

```
net.layers{2}.netInputFcn = 'yournif';
```

Your net input function then is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,

```
N = yournif(Z1,Z2,...)
```

where:

- `Zi` is the ith $S \times Q$ matrix of Q weighted input (column) vectors.
- `N` is an $S \times Q$ matrix of Q net input (column) vectors.

Your net input function must also provide information about itself using this calling format,

```
info = yournif(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'deriv'` - Returns the name of the associated derivative function.

The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.

```
help mynif
z1 = rand(4,5);
z2 = rand(4,5);
z3 = rand(4,5);
n = mynif(z1,z2,z3)
mynif('deriv')
```

Enter the following command to see how `mynif` is implemented.

```
type mynif
```

You can use `mynif` as a template to create your own net input function.

Net Input Derivative Functions. If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,

```
dN_dZ = dtansig(Z,N)
```

where:

- `Z` is one of the $S \times Q$ matrices of Q weighted input (column) vectors.
- `N` is an $S \times Q$ matrix of Q net input (column) vectors.
- `dN_dZ` is the $S \times Q$ derivative d`N`/d`Z`.

To see how the example custom net input derivative function `mydtf` works, type

```
help mydnif
dn_dz1 = mydnif(z1,n)
dn_dz2 = mydnif(z1,n)
dn_dz3 = mydnif(z1,n)
```

Use this command to see how `mydtf` was implemented.

```
type mydnif
```

You can use `mydnif` as a template to create your own net input derivative functions.

Weight Functions

Weight functions calculate a weighted input vector (or matrix) `Z`, given an input vector (or matrices) `P` and a weight matrix `W`.

Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function `yourwf` to the weight going to the second layer from the first input of a network.

```
net.inputWeights{2,1}.weightFcn = 'yourwf';
```

Your weight function is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,

```
Z = yourwf(W,P)
```

where:

- W is an $S \times R$ weight matrix.
- P is an $R \times Q$ matrix of Q input (column) vectors.
- Z is an $S \times Q$ matrix of Q weighted input (column) vectors.

Your net input function must also provide information about itself using this calling format,

```
info = yourwf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.

The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.

```
help mywf
w = rand(1,5);
p = rand(5,1);
z = mywf(w,p);
mywf('deriv')
```

Enter the following command to see how mywf is implemented.

```
type mywf
```

You can use mywf as a template to create your own weight functions.

Weight Derivative Functions. If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,

```
dZ_dP = mydwf('p',W,P,Z)
dZ_dW = mydwf('w',W,P,Z)
```

where:

- W is an $S \times R$ weight matrix.
- P is an $R \times Q$ matrix of Q input (column) vectors.
- Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
- dZ_dP is the $S \times R$ derivative dZ/dP.
- dZ_dW is the $R \times Q$ derivative dZ/dW.

This only works for weight functions whose output consists of a sum of i term, where each ith term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.

To see how the example custom net input derivative function mydwf works, type

```
help mydwf
dz_dp = mydwf('p',w,p,z)
dz_dw = mydwf('w',w,p,z)
```

Use this command to see how mydwf is implemented.

```
type mydwf
```

You can use mydwf as a template to create your own net input derivative function.

Initialization Functions

You can create three kinds of initialization functions: network, layer, and weight/bias initialization.

Network Initialization Functions

The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.

Once defined, you can assign your network initialization function to a network.

```
net.initFcn = 'yournif';
```

Your network initialization function is used whenever you initialize your network.

```
net = init(net)
```

To be a valid network initialization function, it must take and return a network.

```
net = yournif(net)
```

Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.

You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.

Layer Initialization Functions

The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.

Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.

```
net.layers{2}.initFcn = 'yourlif';
```

Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.

```
net = init(net)
```

To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.

```
net = yournif(net,i)
```

Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.

If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.

Weight and Bias Initialization Functions

The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.

Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.

```
net.biases{2}.initFcn = 'yourwbif';
net.inputWeights{2,1}.initFcn = 'yourwbif';
```

Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.

```
net = init(net)
```

To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of R rows defining the minimum and maximum values of R inputs and return a new weight matrix `W`,

```
W = rands(S,PR)
```

where:

- `S` is the number of neurons in the layer.
- `PR` is an $R \times 2$ matrix defining the minimum and maximum values of R inputs.
- `W` is a new $S \times R$ weight matrix.

Your function also needs to generate a new bias vector as follows,

```
b = rands(S)
```

where:

- S is the number of neurons in the layer.
- b is a new $S \times 1$ bias vector.

To see how an example custom weight and bias initialization function works, type

```
help mywbif
W = mywbif(4,[0 1; -2 2])
b = mywbif(4,[1 1])
```

Use this command to see how mywbif was implemented.

```
type mywbif
```

You can use mywbif as a template to create your own weight and bias initialization function.

Learning Functions

You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning.

Training Functions

One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc.

Once defined, you can assign your training function to a network.

```
net.trainFcn = 'yourtf';
```

Your network initialization function is used whenever you train your network.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

To be a valid training function your function must take and return a network,

```
[net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV)
```

where:

- `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
  - Each `Pd{i,j,ts}` is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the *i*th layer from the *j*th input at time step `ts`. (`Pd{i,j,ts}` is an empty matrix `[]` if the *i*th layer doesn't have a weight from the *j*th input.)
- `Tl` is an $N_l \times \mathrm{TS}$ cell array of layer targets.
  - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the *i*th layer. (`Tl{i,ts}` is an empty matrix if the *i*th layer doesn't have a target.)
- `Ai` is an $N_l \times LD$ cell array of initial layer delay states.
  - Each `Ai{l,k}` is the $S^i \times Q$ delayed *i*th layer output for time step `ts` = `k`-*LD*, where ts goes from 0 to *LD*-1.
- `Q` is the number of concurrent vectors.
- `TS` is the number of time steps.
- VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: `Pd`, `Tl`, `Ai`, `Q`, and `TS`. Note that the validation and testing `Q` and `TS` values can be different from each other and from those used by the training vectors.

The dimensions above have the following definitions:

- N_l is the number of network layers (`net.numLayers`).
- N_i is the number of network inputs (`net.numInputs`).
- R^j is the size of the *j*th input (`net.inputs{j}.size`).
- S^i is the size of the *i*th layer (`net.layers{i}.size`)
- *LD* is the number of layer delays (`net.numLayerDelays`).
- D_i^{ij} is the number of delay lines associated with the weight going to the *i*th layer from the *j*th input (`length(net.inputWeights{i,j}.delays)`).

Your training function must also provide information about itself using this calling format,

```
info = yourtf(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'pdefaults'` - Returns a structure of default training parameters.

When you set the network training function (`net.trainFcn`) to be your function, the network's training parameters (`net.trainParam`) automatically is set to your default structure. Those values can be altered (or not) before training.

Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, `train` turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network.

If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as `trainc` and `trainr`. The help for each of these utility functions lists the input and output arguments they take.

Utility Functions. If you examine training functions such as `trainc`, `traingd`, and `trainlm`, note that they use a set of utility functions found in the `nnet/nnutils` directory.

These functions are not listed in Chapter 14, the Reference chapter, because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use `help` on each function to view the function's input and output arguments.

These two functions are useful for creating a new training record and truncating it once the final number of epochs is known:

- `newtr` - New training record with any number of optional fields.
- `cliptr` - Clip training record to the final number of epochs.

These three functions calculate network signals going forward, errors, and derivatives of performance coming back:

- `calca` - Calculate network outputs and other signals.
- `calcerr` - Calculate matrix or cell array errors.
- `calcgrad` - Calculate bias and weight performance gradients.

These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:

- `getx` - Get all network weight and bias values as a single vector.
- `setx` - Set all network weight and bias values with a single vector.

These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:

- `calcperf` - Calculate network outputs, signals, and performance.
- `calcgx` - Calculate weight and bias performance gradient as a single vector.
- `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.

Adapt Functions

The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.

Once defined, you can assign your adapt function to a network.

```
net.adaptFcn = 'youraf';
```

Your network initialization function is used whenever you adapt your network.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid adapt function, it must take and return a network,

```
[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)
```

where:

- `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
  - Each `Pd{i,j,ts}` is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the `i`th layer from the `j`th input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.)

- `Tl` is an $N_l \times \mathrm{TS}$ cell array of layer targets.
  - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the *i*th layer. Note that (`Tl{i,ts}` is an empty matrix if the *i*th layer doesn't have a target.)
- `Ai` is an $N_l \times LD$ cell array of initial layer delay states.
  - Each `Ai{l,k}` is the $S^i \times Q$ delayed *i*th layer output for time step `ts` = `k`-*LD*, where ts goes from 0 to *LD*-1.
- `Q` is the number of concurrent vectors.
- `TS` is the number of time steps.

The dimensions above have the following definitions:

- N_l is the number of network layers (`net.numLayers`).
- N_i is the number of network inputs (`net.numInputs`).
- R^j is the size of the jth input (`net.inputs{j}.size`).
- S^i is the size of the ith layer (`net.layers{i}.size`)
- *LD* is the number of layer delays (`net.numLayerDelays`).
- D_i^{ij} is the number of delay lines associated with the weight going to the *i*th layer from the *j*th input (`length(net.inputWeights{i,j}.delays)`).

Your adapt function must also provide information about itself using this calling format,

```
info = youraf(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'pdefaults'` - Returns a structure of default adapt parameters.

When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting.

Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network.

If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.

Utility Functions. If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.

These functions are not listed in Chapter 14 because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.

These two functions are useful for simulating a network, and calculating its derivatives of performance:

- `calca1` - New training record with any number of optional fields.
- `calce1` - Clip training record to the final number of epochs.
- `calcgrad` - Calculate bias and weight performance gradients.

Performance Functions

Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.

Once defined you can assign your training function to a network.

```
net.performFcn = 'yourpf';
```

Your network initialization function will then be used whenever you train your adapt your network.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid performance function your function must be called as follows,

```
perf = yourpf(E,X,PP)
```

where:

- `E` is either an S x Q matrix or an $N_l \times \mathrm{TS}$ cell array of layer errors.
  - Each `E{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl(i,ts)` is an empty matrix if the ith layer doesn't have a target.)

- `X` is an $M \times 1$ vector of all the network's weights and biases.
- `PP` is a structure of network performance parameters.

If E is a cell array you can convert it to a matrix as follows.

```
E = cell2mat(E);
```

Alternatively, your function must also be able to be called as follows,

```
perf = yourpf(E,net)
```

where you can get `X` and `PP` (if needed) as follows.

```
X = getx(net);
PP = net.performParam;
```

Your performance function must also provide information about itself using this calling format,

```
info = yourpf(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'deriv'` - Returns the name of the associated derivative function.
- `'pdefaults'` - Returns a structure of default performance parameters.

When you set the network performance function (`net.performFcn`) to be your function, the network's adapt parameters (`net.performParam`) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.

To see how an example custom performance function works type in these lines of code.

```
help mypf
e = rand(4,5);
x = rand(12,1);
pp = mypf('pdefaults')
perf = mypf(e,x,pp)
```

Use this command to see how `mypf` was implemented.

```
type mypf
```

You can use mypf as a template to create your own weight and bias initialization function.

Performance Derivative Functions. If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,

```
dPerf_dE = dmsereg('e',E,X,perf,PP)
dPerf_dX = dmsereg('x',E,X,perf,PP)
```

where:

- E is an $N_l \times \mathrm{TS}$ cell array of layer errors.
  - Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the *i*th layer doesn't have a target.)
- X is an $M \times 1$ vector of all the network's weights and biases.
- PP is a structure of network performance parameters.
- dPerf_dE is the $N_l \times \mathrm{TS}$ cell array of derivatives *d*Perf/*d*E.
  - Each E{i,ts} is the $S^i \times Q$ derivative matrix for the *i*th layer. Note that (Tl(i,ts) is an empty matrix if the *i*th layer doesn't have a target.)
- dPerf_dX is the $M \times 1$ derivative *d*Perf/*d*X.

To see how the example custom performance derivative function mydpf works, type

```
help mydpf
e = {e};
dperf_de = mydpf('e',e,x,perf,pp)
dperf_dx = mydpf('x',e,x,perf,pp)
```

Use this command to see how mydpf was implemented.

```
type mydpf
```

You can use mydpf as a template to create your own performance derivative functions.

Weight and Bias Learning Functions

The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.

Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.

```
net.biases{2}.learnFcn = 'yourwblf';
net.inputWeights{2,1}.learnFcn = 'yourwblf';
```

Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid weight and bias learning function, it must be callable as follows,

```
[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
```

where:

- `W` is an $S \times R$ weight matrix.
- `P` is an $R \times Q$ matrix of Q input (column) vectors.
- `Z` is an $S \times Q$ matrix of Q weighted input (column) vectors.
- `N` is an $S \times Q$ matrix of Q net input (column) vectors.
- `A` is an $S \times Q$ matrix of Q layer output (column) vectors.
- `T` is an $S \times Q$ matrix of Q target (column) vectors.
- `E` is an $S \times Q$ matrix of Q error (column) vectors.
- `gW` is an $S \times R$ gradient of `W` with respect to performance.
- `gA` is an $S \times Q$ gradient of `A` with respect to performance.
- `D` is an $S \times S$ matrix of neuron distances.
- `LP` is a a structure of learning parameters.

- LS is a structure of the learning state that is updated for each call. (Use a null matrix [] the first time.)
- dW is the resulting $S \times R$ weight change matrix.

Your function is called as follows to update bias vector

```
[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
```

where:

- S is the number of neurons in the layer.
- b is a new $S \times 1$ bias vector.

Your learning function must also provide information about itself using this calling format,

```
info = yourwblf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.
- 'pdefaults' - Returns a structure of default performance parameters.

To see how an example custom weight and bias initialization function works, type

```
help mywblf
```

Use this command to see how mywbif was implemented.

```
type mywblf
```

You can use mywblf as a template to create your own weight and bias learning function.

Self-Organizing Map Functions

There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.

Topology Functions

Topology functions calculate the positions of a layer's neurons given its dimensions.

Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function `yourtopf` to the second layer of a network.

```
net.layers{2}.topologyFcn = 'yourtopf';
```

Your topology function is used whenever your network is trained or adapts.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid topology function your function must calculate positions `pos` from dimensions `dim` as follows,

```
pos = yourtopf(dim1,dim2,...,dimN)
```

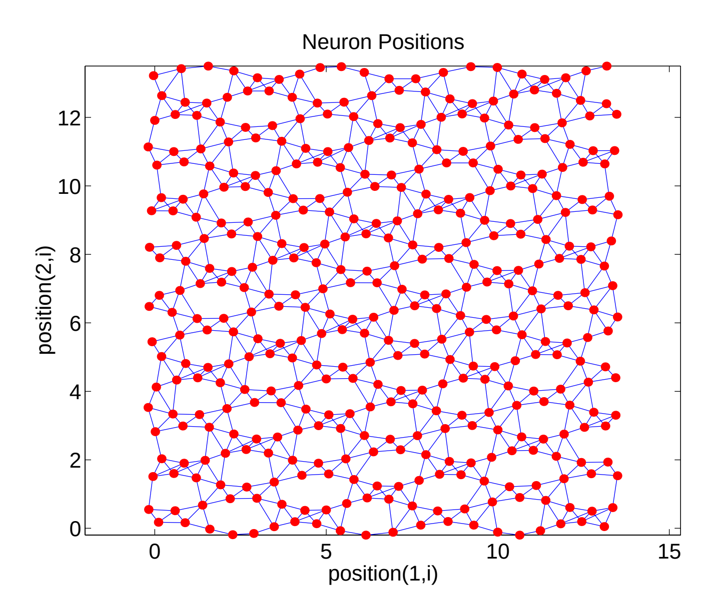
where:

- `dimi` is the number of neurons along the ith dimension of the layer.
- `pos` is an $N \times S$ matrix of S position vectors, where S is the total number of neurons that is defined by the product `dim1*dim1*...*dimN`.

The toolbox contains an example custom topology function called `mytopf`. Enter the following lines of code to see how it is used.

```
help mytopf
pos = mytopf(20,20);
plotsom(pos)
```

If you type that code, you get the following plot.



Enter the following command to see how `mytf` is implemented.

```
type mytopf
```

You can use `mytopf` as a template to create your own topology function.

Distance Functions

Distance functions calculate the distances of a layer's neurons given their position.

Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function `yourdistf` to the second layer of a network.

```
net.layers{2}.distanceFcn = 'yourdistf';
```

Your distance function is used whenever your network is trained or adapts.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid distance function, it must calculate distances `d` from position `pos` as follows,

```
pos = yourtopf(dim1,dim2,...,dimN)
```

where:

- `pos` is an $N \times S$ matrix of S neuron position vectors.
- `d` is an $S \times S$ matrix of neuron distances.

The toolbox contains an example custom distance function called `mydistf`. Enter the following lines of code to see how it is used.

```
help mydistf
pos = gridtop(4,5);
d = mydistf(pos)
```

Enter the following command to see how `mytf` is implemented.

```
type mydistf
```

You can use `mydistf` as a template to create your own distance function.

13

Network Object Reference

Network Properties 13-2
Architecture 13-2
Subobject Structures 13-6
Functions 13-9
Parameters 13-12
Weight and Bias Values 13-14
Other 13-16

Subobject Properties 13-17
Inputs 13-17
Layers 13-18
Outputs 13-25
Targets 13-25
Biases 13-26
Input Weights 13-28
Layer Weights 13-32

Network Properties

The properties define the basic features of a network. A later section, “Subobject Properties” describes properties that define network details.

Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

numInputs

This property defines the number of inputs a network receives.

```
net.numInputs
```

It can be set to 0 or a positive integer.

Clarification. The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e. the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

Side Effects. Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

numLayers

This property defines the number of layers a network has.

```
net.numLayers
```

It can be set to 0 or a positive integer.

Side Effects. Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers,

```
net.biasConnect
net.inputConnect
net.layerConnect
```

```
net.outputConnect
net.targetConnect
```

and changes the size each cell array of subobject structures whose size depends on the number of layers,

```
net.biases
net.inputWeights
net.layerWeights
net.outputs
net.targets
```

and also changes the size of each of the network's adjustable parameters properties.

```
net.IW
net.LW
net.b
```

biasConnect

This property defines which layers have biases.

```
net.biasConnect
```

It can be set to any N-by-1 matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the ith layer is indicated by a 1 (or 0) at:

```
net.biasConnect(i)
```

Side Effects. Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

inputConnect

This property defines which layers have weights coming from inputs.

```
net.inputConnect
```

It can be set to any $N_l \times N_i$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the ith layer from the jth input is indicated by a 1 (or 0) at:

```
net.inputConnect(i,j)
```

Side Effects. Any change to this property will alter the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and in the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

layerConnect

This property defines which layers have weights coming from other layers.

```
net.layerConnect
```

It can be set to any $N_l \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a weight going to the ith layer from the jth layer is indicated by a 1 (or 0) at:

```
net.layerConnect(i,j)
```

Side Effects. Any change to this property will alter the presence or absence of structures in the cell array of layer weight subobjects (`net.layerWeights`) and in the presence or absence of matrices in the cell array of layer weight matrices (`net.LW`).

outputConnect

This property defines which layers generate network outputs.

```
net.outputConnect
```

It can be set to any $1 \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a network output from the ith layer is indicated by a 1 (or 0) at:

```
net.outputConnect(i)
```

Side Effects. Any change to this property will alter the number of network outputs (`net.numOutputs`) and the presence or absence of structures in the cell array of output subobjects (`net.outputs`).

targetConnect

This property defines which layers have associated targets.

```
net.targetConnect
```

It can be set to any $1 \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a target associated with the ith layer is indicated by a 1 (or 0) at:

```
net.targetConnect(i)
```

Side Effects. Any change to this property alters the number of network targets (`net.numTargets`) and the presence or absence of structures in the cell array of target subobjects (`net.targets`).

numOutputs (read-only)

This property indicates how many outputs the network has.

```
net.numOutputs
```

It is always set to the number of 1's in the matrix of output connections.

```
numOutputs = sum(net.outputConnect)
```

numTargets (read-only)

This property indicates how many targets the network has.

```
net.numTargets
```

It is always set to the number of 1's in the matrix of target connections.

```
numTargets = sum(net.targetConnect)
```

numInputDelays (read-only)

This property indicates the number of time steps of past inputs that must be supplied to simulate the network.

```
net.numInputDelays
```

It is always set to the maximum delay value associated any of the network's input weights.

```
numInputDelays = 0;
for i=1:net.numLayers
  for j=1:net.numInputs
    if net.inputConnect(i,j)
      numInputDelays = max( ...
        [numInputDelays net.inputWeights{i,j}.delays]);
```

```
    end
  end
end
```

numLayerDelays (read-only)

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network.

```
net.numLayerDelays
```

It is always set to the maximum delay value associated any of the network's layer weights.

```
numLayerDelays = 0;
for i=1:net.numLayers
  for j=1:net.numLayers
    if net.layerConnect(i,j)
      numLayerDelays = max( ...
       [numLayerDelays net.layerWeights{i,j}.delays]);
    end
  end
end
```

Subobject Structures

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

The properties for each kind of subobject are described in the "Subobject Properties" section that follows this "Network Properties" section.

inputs

This property holds structures of properties for each of the network's inputs.

```
net.inputs
```

It is always an $N_i \times 1$ cell array of input structures, where N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the ith network input is located at:

```
net.inputs{i}
```

Input Properties. See “Inputs” in the “Subobject Properties” section for descriptions of input properties.

layers

This property holds structures of properties for each of the network’s layers.

```
net.layers
```

It is always an $N_l \times 1$ cell array of layer structures, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the ith layer is located at:

```
net.layers{i}
```

Layer Properties. See “Layers” in the “Subobject Properties” section for descriptions of layer properties.

outputs

This property holds structures of properties for each of the network’s outputs.

```
net.outputs
```

It is always an $1 \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the output from the ith layer (or a null matrix `[]`) is located at

```
net.outputs{i}
```

if the corresponding output connection is 1 (or 0).

```
net.outputConnect(i)
```

Output Properties. See “Outputs” in the “Subobject Properties” section for descriptions of output properties.

targets

This property holds structures of properties for each of the network’s targets.

```
net.targets
```

It is always an $1 \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the target associated with the ith layer (or a null matrix `[]`) is located at

```
net.targets{i}
```

if the corresponding target connection is 1 (or 0).

```
net.targetConnect(i)
```

Target Properties. See “Targets” in the “Subobject Properties” section for descriptions of target properties.

biases

This property holds structures of properties for each of the network’s biases.

```
net.biases
```

It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the ith layer (or a null matrix `[]`) is located at

```
net.biases{i}
```

if the corresponding bias connection is 1 (or 0).

```
net.biasConnect(i)
```

Bias Properties. See “Biases” in the “Subobject Properties” section for descriptions of bias properties.

inputWeights

This property holds structures of properties for each of the network’s input weights.

```
net.inputWeights
```

It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the *i*th layer from the *j*th input (or a null matrix `[]`) is located at

```
net.inputWeights{i,j}
```

if the corresponding input connection is 1 (or 0).

```
net.inputConnect(i,j)
```

Input Weight Properties. See "Input Weights" in the "Subobject Properties" section for descriptions of input weight properties.

layerWeights

This property holds structures of properties for each of the network's layer weights.

```
net.layerWeights
```

It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the weight going to the *i*th layer from the *j*th layer (or a null matrix `[]`) is located at:

```
net.layerWeights{i,j}
```

if the corresponding layer connection is 1 (or 0).

```
net.layerConnect(i,j)
```

Layer Weight Properties. See "Layer Weights" in the "Subobject Properties" section for descriptions of layer weight properties.

Functions

These properties define the algorithms to use when a network is to adapt, is to be initialized, is to have its performance measured, or is to be trained.

adaptFcn

This property defines the function to be used when the network adapts.

```
net.adaptFcn
```

It can be set to the name of any network adapt function, including this toolbox function:

```
trains - By-weight-and-bias network adaption function.
```

The network adapt function is used to perform adaption whenever `adapt` is called.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

Custom Functions. See Chapter 12 for information on creating custom adapt functions.

Side Effects. Whenever this property is altered, the network's adaption parameters (`net.adaptParam`) are set to contain the parameters and default values of the new function.

initFcn

This property defines the function used to initialize the network's weight matrices and bias vectors.

```
net.initFcn
```

It can be set to the name of any network initialization function, including this toolbox function.

```
initlay  - Layer-by-layer network initialization function.
```

The initialization function is used to initialize the network whenever `init` is called.

```
net = init(net)
```

Custom Functions. See Chapter 12 for information on creating custom initialization functions.

Side Effects. Whenever this property is altered, the network's initialization parameters (`net.initParam`) are set to contain the parameters and default values of the new function.

performFcn

This property defines the function used to measure the network's performance.

```
net.performFcn
```

It can be set to the name of any performance function, including these toolbox functions.

Performance Functions	
mae	Mean absolute error-performance function.
mse	Mean squared error-performance function.
msereg	Mean squared error w/reg performance function.
sse	Sum squared error-performance function.

The performance function is used to calculate network performance during training whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

Custom functions. See "Advanced Topics" in Chapter 12 for information on creating custom performance functions.

Side Effects. Whenever this property is altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

trainFcn

This property defines the function used to train the network.

```
net.trainFcn
```

It can be set to the name of any training function, including these toolbox functions.

Training Functions	
trainbfg	BFGS quasi-Newton backpropagation.
trainbr	Bayesian regularization.
traincgb	Powell-Beale conjugate gradient backpropagation.

Training Functions	
traincgf	Fletcher-Powell conjugate gradient backpropagation.
traincgp	Polak-Ribiere conjugate gradient backpropagation.
traingd	Gradient descent backpropagation.
traingda	Gradient descent with adaptive lr backpropagation.
traingdm	Gradient descent with momentum backpropagation.
traingdx	Gradient descent with momentum and adaptive lr backprop.
trainlm	Levenberg-Marquardt backpropagation.
trainoss	One-step secant backpropagation.
trainrp	Resilient backpropagation (Rprop).
trainscg	Scaled conjugate gradient backpropagation.
trainb	Batch training with weight and bias learning rules.
trainc	Cyclical order incremental training with learning functions.
trainr	Random order incremental training with learning functions.

The training function is used to train the network whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

Custom Functions. See "Advanced Topics" in Chapter 12 for information on creating custom training functions.

Side Effects. Whenever this property is altered, the network's training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

Parameters

adaptParam

This property defines the parameters and values of the current adapt function.

```
net.adaptParam
```

The fields of this property depend on the current adapt function (`net.adaptFcn`). Evaluate the above reference to see the fields of the current adapt function.

Call `help` on the current adapt function to get a description of what each field means.

```
help(net.adaptFcn)
```

initParam

This property defines the parameters and values of the current initialization function.

```
net.initParam
```

The fields of this property depend on the current initialization function (`net.initFcn`). Evaluate the above reference to see the fields of the current initialization function.

Call `help` on the current initialization function to get a description of what each field means.

```
help(net.initFcn)
```

performParam

This property defines the parameters and values of the current performance function.

```
net.performParam
```

The fields of this property depend on the current performance function (`net.performFcn`). Evaluate the above reference to see the fields of the current performance function.

Call `help` on the current performance function to get a description of what each field means.

```
help(net.performFcn)
```

trainParam

This property defines the parameters and values of the current training function.

```
net.trainParam
```

The fields of this property depend on the current training function (`net.trainFcn`). Evaluate the above reference to see the fields of the current training function.

Call `help` on the current training function to get a description of what each field means.

```
help(net.trainFcn)
```

Weight and Bias Values

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

IW

This property defines the weight matrices of weights going to layers from network inputs.

```
net.IW
```

It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The weight matrix for the weight going to the *i*th layer from the *j*th input (or a null matrix `[]`) is located at

```
net.IW{i,j}
```

if the corresponding input connection is 1 (or 0).

```
net.inputConnect(i,j)
```

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the input size with the number of delays associated with the weight.

```
net.inputs{j}.size * length(net.inputWeights{i,j}.delays)
```

These dimensions can also be obtained from the input weight properties.

```
net.inputWeights{i,j}.size
```

LW

This property defines the weight matrices of weights going to layers from other layers.

```
net.LW
```

It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The weight matrix for the weight going to the *i*th layer from the *j*th layer (or a null matrix `[]`) is located at

```
net.LW{i,j}
```

if the corresponding layer connection is 1 (or 0).

```
net.layerConnect(i,j)
```

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight.

```
net.layers{j}.size * length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties.

```
net.layerWeights{i,j}.size
```

b

This property defines the bias vectors for each layer with a bias.

```
net.b
```

It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The bias vector for the *i*th layer (or a null matrix `[]`) is located at

```
net.b{i}
```

if the corresponding bias connection is 1 (or 0).

```
net.biasConnect(i)
```

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties.

```
net.biases{i}.size
```

Other

The only `other` property is a user data property.

userdata

This property provides a place for users to add custom information to a network object.

```
net.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.userdata.note
```

Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

Inputs

These properties define the details of each *i*th network input.

```
net.inputs{i}
```

range

This property defines the ranges of each element of the *i*th network input.

```
net.inputs{i}.range
```

It can be set to any $R_i \times 2$ matrix, where R_i is the number of elements in the input (`net.inputs{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum values of the *j*th input element, in that order:

```
net.inputs{i}(j,:)
```

Uses. Some initialization functions use input ranges to find appropriate initial values for input weight matrices.

Side Effects. Whenever the number of rows in this property is altered, the layers's size (`net.inputs{i}.size`) changes to remain consistent. The size of any weights coming from this input (`net.inputWeights{:,i}.size`) and the dimensions of their weight matrices (`net.IW{:,i}`) also changes size.

size

This property defines the number of elements in the *i*th network input.

```
net.inputs{i}.size
```

It can be set to 0 or a positive integer.

Side Effects. Whenever this property is altered, the input's ranges (`net.inputs{i}.ranges`), any input weights (`net.inputWeights{:,i}.size`) and their weight matrices (`net.IW{:,i}`) change size to remain consistent.

userdata

This property provides a place for users to add custom information to the *i*th network input.

```
net.inputs{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.inputs{i}.userdata.note
```

Layers

These properties define the details of each *i*th network layer.

```
net.layers{i}
```

dimensions

This property defines the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

```
net.layers{i}.dimensions
```

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements will becomes the number of neurons in the layer (`net.layers{i}.size`).

Uses. Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

Side Effects. Whenever this property is altered, the layers's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

distanceFcn

This property defines the function used to calculate distances between neurons in the *i*th layer (`net.layers{i}.distances`) from the neuron positions (`net.layers{i}.positions`). Neuron distances are used by self-organizing maps.

```
net.layers{i}.distanceFcn
```

It can be set to the name of any distance function, including these toolbox functions.

Distance Functions	
boxdist	Distance between two position vectors.
dist	Euclidean distance weight function.
linkdist	Link distance function.
mandist	Manhattan distance weight function.

Custom Functions. See “Advanced Topics” in Chapter 12 for information on creating custom distance functions.

Side Effects. Whenever this property is altered, the distance between the layer’s neurons (`net.layers{i}.distances`) is updated.

distances (read-only)

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps.

```
net.layers{i}.distances
```

It is always set to the result of applying the layer’s distance function (`net.layers{i}.distanceFcn`) to the positions of the layers neurons (`net.layers{i}.positions`).

initFcn

This property defines the initialization function used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is `initlay`.

```
net.layers{i}.initFcn
```

It can be set to the name of any layer initialization function, including these toolbox functions.

Layer Initialization Functions	
`initnw`	Nguyen-Widrow layer initialization function.
`initwb`	By-weight-and-bias layer initialization function.

If the network initialization is set to `initlay`, then the function indicated by this property is used to initialize the layer's weights and biases when `init` is called.

```
net = init(net)
```

Custom Functions. See Chapter 12 for information on creating custom initialization functions.

netInputFcn

This property defines the net input function use to calculate the *i*th layer's net input, given the layer's weighted inputs and bias.

```
net.layers{i}.netInputFcn
```

It can be set to the name of any net input function, including these toolbox functions.

Net Input Functions	
`netprod`	Product net input function.
`netsum`	Sum net input function.

The net input function is used to simulate the network when `sim` is called.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

Custom Functions. See Chapter 12 for information on creating custom net input functions.

positions (read-only)

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

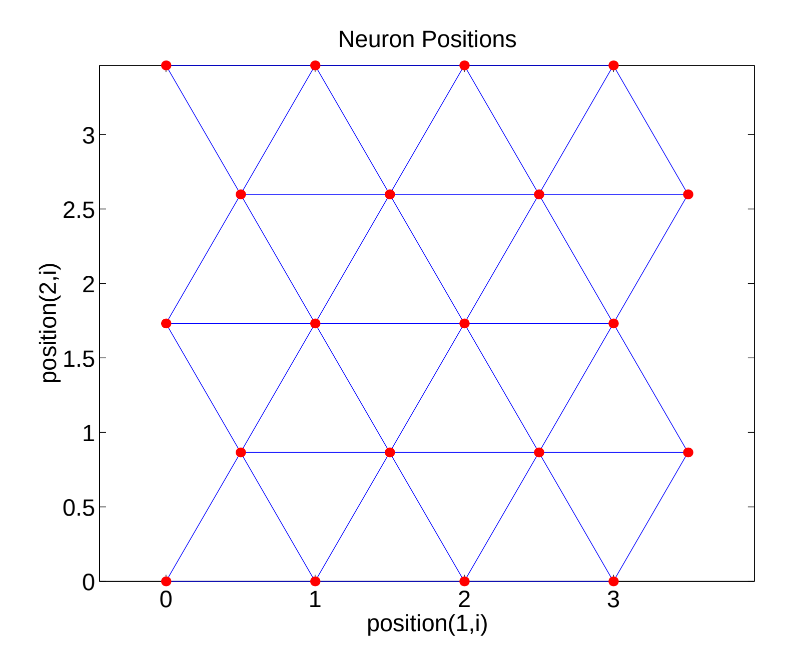
```
net.layers{i}.positions
```

It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

Plotting. Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5] and the topology function (`net.layers{1}.topologyFcn`) is `hextop`, the neuron's positions can be plotted as shown below.

```
plotsom(net.layers{1}.positions)
```



size

This property defines the number of neurons in the *i*th layer.

```
net.layers{i}.size
```

It can be set to 0 or a positive integer.

Side Effects. Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), and any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.inputWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`) change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`) and biases (`net.b{i}`) also change.

Changing this property also changes the size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the `dimensions` property directly instead of using `size`).

topologyFcn

This property defines the function used to calculate the *ith* layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

```
net.topologyFcn
```

It can be set to the name of any topology function, including these toolbox functions.

Topology Functions	
`gridtop`	Gridtop layer topology function.
`hextop`	Hexagonal layer topology function.
`randtop`	Random layer topology function.

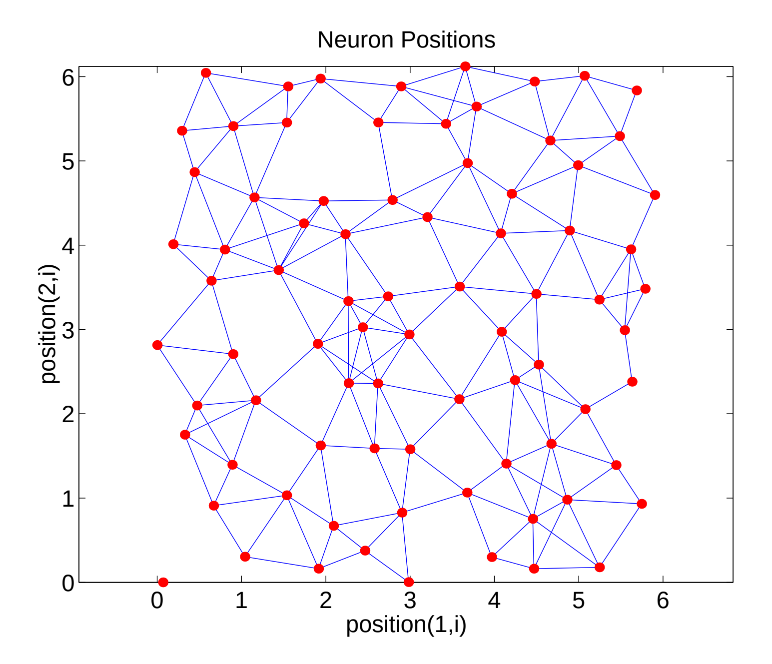
Custom functions. See Chapter 12 for information on creating custom topology functions.

Side Effects. Whenever this property is altered, the positions of the layer's neurons (`net.layers{i}.positions`) is updated.

Plotting. Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is `randtop`, the neuron's positions are arranged something like those shown in the plot below.

```
plotsom(net.layers{1}.positions)
```



transferFcn

This function defines the transfer function used to calculate the ith layer's output, given the layer's net input.

```
net.layers{i}.transferFcn
```

It can be set to the name of any transfer function, including these toolbox functions.

Transfer Functions	
compet	Competitive transfer function.
hardlim	Hard-limit transfer function.
hardlims	Symmetric hard-limit transfer function.
logsig	Log-sigmoid transfer function.
poslin	Positive linear transfer function.
purelin	Hard-limit transfer function.
radbas	Radial basis transfer function.
satlin	Saturating linear transfer function.
satlins	Symmetric saturating linear transfer function.
softmax	Soft max transfer function.
tansig	Hyperbolic tangent sigmoid transfer function.
tribas	Triangular basis transfer function.

The transfer function is used to simulate the network when `sim` is called.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

Custom functions. See Chapter 12 for information on creating custom transfer functions.

userdata

This property provides a place for users to add custom information to the *i*th network layer.

```
net.layers{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.layers{i}.userdata.note
```

Outputs

size (read-only)

This property defines the number of elements in the *i*th layer's output.

```
net.outputs{i}.size
```

It is always set to the size of the *ith* layer (`net.layers{i}.size`).

userdata

This property provides a place for users to add custom information to the *i*th layer's output.

```
net.outputs{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.outputs{i}.userdata.note
```

Targets

size (read-only)

This property defines the number of elements in the *i*th layer's target.

```
net.targets{i}.size
```

It is always set to the size of the *ith* layer (`net.layers{i}.size`).

userdata

This property provides a place for users to add custom information to the *i*th layer's target.

```
net.targets{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.targets{i}.userdata.note
```

Biases

initFcn

This property defines the function used to initialize the *i*th layer's bias vector, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.biases{i}.initFcn
```

This function can be set to the name of any bias initialization function, including the toolbox functions.

Bias Initialization Functions	
`initcon`	Conscience bias initialization function.
`initzero`	Zero-weight/bias initialization function.
`rands`	Symmetric random weight/bias initialization function.

This function is used to calculate an initial bias vector for the *i*th layer (`net.b{i}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *ith* layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

Custom functions. See Chapter 12 for information on creating custom initialization functions.

learn

This property defines whether the *i*th bias vector is to be altered during training and adaption.

```
net.biases{i}.learn
```

It can be set to 0 or 1.

It enables or disables the bias' learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

learnFcn

This property defines the function used to update the *i*th layer's bias vector during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.biases{i}.learnFcn
```

It can be set to the name of any bias learning function, including these toolbox functions.

Learning Functions	
`learncon`	Conscience bias learning function.
`learngd`	Gradient descent weight/bias learning function.
`learngdm`	Grad. descent w/momentum weight/bias learning function.
`learnp`	Perceptron weight/bias learning function.
`learnpn`	Normalized perceptron weight/bias learning function.
`learnwh`	Widrow-Hoff weight/bias learning rule.

The learning function updates the *ith* bias vector (`net.b{i}`) during calls to `train`, if the network training function (`net.trainFcn`) is `trainb`, `trainc`, or `trainr`, or during calls to `adapt`, if the network adapt function (`net.adaptFcn`) is `trains`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

Custom functions. See Chapter 12 for information on creating custom learning functions.

Side Effects. Whenever this property is altered, the biases's learning parameters (`net.biases{i}.learnParam`) are set to contain the fields and default values of the new function.

learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias.

```
net.biases{i}.learnParam
```

The fields of this property depend on the current learning function (`net.biases{i}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

```
help(net.biases{i}.learnFcn)
```

size (read-only)

This property defines the size of the *i*th layer's bias vector.

```
net.biases{i}.size
```

It is always set to the size of the *i*th layer (`net.layers{i}.size`).

userdata

This property provides a place for users to add custom information to the *i*th layer's bias.

```
net.biases{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.biases{i}.userdata.note
```

Input Weights

delays

This property defines a tapped delay line between the *j*th input and its weight to the *i*th layer.

```
net.inputWeights{i,j}.delays
```

It must be set to a row vector of increasing 0 or positive integer values.

Side Effects. Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

initFcn

This property defines the function used to initialize the weight matrix going to the *i*th layer from the *j*th input, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.inputWeights{i,j}.initFcn
```

This function can be set to the name of any weight initialization function, including these toolbox functions.

Weight Initialization Functions	
`initzero`	Zero-weight/bias initialization function.
`midpoint`	Midpoint-weight initialization function.
`randnc`	Normalized column-weight initialization function.
`randnr`	Normalized row-weight initialization function.
`rands`	Symmetric random-weight/bias initialization function.

This function is used to calculate an initial weight matrix for the weight going to the *i*th layer from the *j*th input (`net.IW{i,j}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *i*th layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

Custom Functions. See Chapter 12 for information on creating custom initialization functions.

learn

This property defines whether the weight matrix to the *i*th layer from the *j*th input is to be altered during training and adaption.

```
net.inputWeights{i,j}.learn
```

It can be set to 0 or 1.

It enables or disables the weights learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

learnFcn

This property defines the function used to update the weight matrix going to the *i*th layer from the *j*th input during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.inputWeights{i,j}.learnFcn
```

It can be set to the name of any weight learning function, including these toolbox functions.

Weight Learning Functions	
`learngd`	Gradient descent weight/bias learning function.
`learngdm`	Grad. descent w/ momentum weight/bias learning function.
`learnh`	Hebb-weight learning function.
`learnhd`	Hebb with decay weight learning function.
`learnis`	Instar-weight learning function.
`learnk`	Kohonen-weight learning function.
`learnlv1`	LVQ1-weight learning function.
`learnlv2`	LVQ2-weight learning function.
`learnos`	Outstar-weight learning function.
`learnp`	Perceptron weight/bias learning function.
`learnpn`	Normalized perceptron-weight/bias learning function.
`learnsom`	Self-organizing map-weight learning function.
`learnwh`	Widrow-Hoff weight/bias learning rule.

The learning function updates the weight matrix of the *i*th layer from the *j*th input (`net.IW{i,j}`) during calls to `train`, if the network training function

(`net.trainFcn`) is `trainb`, `trainc`, or `trainr`, or during calls to `adapt`, if the network adapt function (`net.adaptFcn`) is `trains`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

Custom Functions. See Chapter 12 for information on creating custom learning functions.

learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's weight coming from the *j*th input.

```
net.inputWeights{i,j}.learnParam
```

The fields of this property depend on the current learning function (`net.inputWeights{i,j}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

```
help(net.inputWeights{i,j}.learnFcn)
```

size (read-only)

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th network input.

```
net.inputWeights{i,j}.size
```

It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.IW{i,j}`). The first element is equal to the size of the *i*th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weights delay vectors with the size of the *j*th input:

```
length(net.inputWeights{i,j}.delays) * net.inputs{j}.size
```

userdata

This property provides a place for users to add custom information to the *(i,j)*th input weight.

```
net.inputWeights{i,j}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.inputWeights{i,j}.userdata.note
```

weightFcn

This property defines the function used to apply the *i*th layer's weight from the *j*th input to that input.

```
net.inputWeights{i,j}.weightFcn
```

It can be set to the name of any weight function, including these toolbox functions.

Weight Functions	
`dist`	Conscience bias initialization function.
`dotprod`	Zero-weight/bias initialization function.
`mandist`	Manhattan-distance weight function.
`negdist`	Normalized column-weight initialization function.
`normprod`	Normalized row-weight initialization function.

The weight function is used when `sim` is called to simulate the network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

Custom functions. See Chapter 12 for information on creating custom weight functions.

Layer Weights

delays

This property defines a tapped delay line between the *j*th layer and its weight to the *ith* layer.

```
net.layerWeights{i,j}.delays
```

It must be set to a row vector of increasing 0 or positive integer values.

initFcn

This property defines the function used to initialize the weight matrix going to the *i*th layer from the *j*th layer, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.layerWeights{i,j}.initFcn
```

This function can be set to the name of any weight initialization function, including the toolbox functions.

Weight and Bias Initialization Functions	
`initzero`	Zero-weight/bias initialization function.
`midpoint`	Midpoint-weight initialization function.
`randnc`	Normalized column-weight initialization function.
`randnr`	Normalized row-weight initialization function.
`rands`	Symmetric random-weight/bias initialization function.

This function is used to calculate an initial weight matrix for the weight going to the *i*th layer from the *j*th layer (`net.LW{i,j}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *i*th layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

Custom Functions. See Chapter 12 for information on creating custom initialization functions.

learn

This property defines whether the weight matrix to the *ith* layer from the *j*th layer is to be altered during training and adaption.

```
net.layerWeights{i,j}.learn
```

It can be set to 0 or 1.

It enables or disables the weights learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

learnFcn

This property defines the function used to update the weight matrix going to the *i*th layer from the *j*th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.layerWeights{i,j}.learnFcn
```

It can be set to the name of any weight learning function, including these toolbox functions.

Learning Functions	
`learngd`	Gradient-descent weight/bias learning function.
`learngdm`	Grad. descent w/momentum weight/bias learning function.
`learnh`	Hebb-weight learning function.
`learnhd`	Hebb with decay weight learning function.
`learnis`	Instar-weight learning function.
`learnk`	Kohonen-weight learning function.
`learnlv1`	LVQ1-weight learning function.
`learnlv2`	LVQ2-weight learning function.
`learnos`	Outstar-weight learning function.
`learnp`	Perceptron-weight/bias learning function.
`learnpn`	Normalized perceptron-weight/bias learning function.
`learnsom`	Self-organizing map-weight learning function.
`learnwh`	Widrow-Hoff weight/bias learning rule.

The learning function updates the weight matrix of the *i*th layer form the *j*th layer (`net.LW{i,j}`) during calls to `train`, if the network training function

(net.trainFcn) is trainb, trainc, or trainr, or during calls to adapt, if the network adapt function (net.adaptFcn) is trains.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

Custom Functions. See Chapter 12 for information on creating custom learning functions.

learnParam

This property defines the learning parameters fields and values for the current learning function of the *ith* layer's weight coming from the *j*th layer.

```
net.layerWeights{i,j}.learnParam
```

The subfields of this property depend on the current learning function (net.layerWeights{i,j}.learnFcn). Evaluate the above reference to see the fields of the current learning function.

Get help on the current learning function to get a description of what each field means.

```
help(net.layerWeights{i,j}.learnFcn)
```

size (read-only)

This property defines the dimensions of the *ith* layer's weight matrix from the *j*th layer.

```
net.layerWeights{i,j}.size
```

It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (net.LW{i,j}). The first element is equal to the size of the *ith* layer (net.layers{i}.size). The second element is equal to the product of the length of the weights delay vectors with the size of the *j*th layer.

```
length(net.layerWeights{i,j}.delays) * net.layers{j}.size
```

userdata

This property provides a place for users to add custom information to the *(i,j)th* layer weight.

```
net.layerWeights{i,j}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.layerWeights{i,j}.userdata.note
```

weightFcn

This property defines the function used to apply the *i*th layer's weight from the *j*th layer to that layer's output.

```
net.layerWeights{i,j}.weightFcn
```

It can be set to the name of any weight function, including these toolbox functions.

Weight Functions	
dist	Euclidean-distance weight function.
dotprod	Dot-product weight function.
mandist	Manhattan-distance weight function.
negdist	Dot-product weight function.
normprod	Normalized dot-product weight function.

The weight function is used when `sim` is called to simulate the network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

Custom Functions. See Chapter 12 for information on creating custom weight functions.

14

Reference

Functions — By Category 14-2
Functions by Network Type 14-2
Functions by Class 14-2
Transfer Functions 14-13
Transfer Function Graphs 14-14

Functions — Alphabetical List 14-19
Reference Headings 14-19

Functions — By Category

Functions by Network Type

Function and Network Type	
assoclr	Associative learning rules
backprop	Backpropagation networks
elman	Elman recurrent networks
hopfield	Hopfield recurrent networks
linnet	Linear networks
lvq	Learning vector quantization
percept	Perceptrons
radbasis	Radial basis networks
selforg	Self-organizing networks

Functions by Class

Analysis Functions	
errsurf	Error surface of a single input neuron.
maxlinlr	Maximum learning rate for a linear neuron.

Distance Functions	
`boxdist`	Distance between two position vectors.
`dist`	Euclidean distance weight function.
`linkdist`	Link distance function.
`mandist`	Manhattan distance weight function.

Graphical Interface Function	
`nntool`	Neural Network Tool - Graphical User Interface.

Layer Initialization Functions	
`initnw`	Nguyen-Widrow layer initialization function.
`initwb`	By-weight-and-bias layer initialization function.

Learning Functions	
`learncon`	Conscience bias learning function.
`learngd`	Gradient descent weight/bias learning function.
`learngdm`	Grad. descent w/momentum weight/bias learning function.
`learnh`	Hebb weight learning function.
`learnhd`	Hebb with decay weight learning rule.
`learnis`	Instar weight learning function.

Learning Functions	
learnk	Kohonen weight learning function.
learnlv1	LVQ1 weight learning function.
learnlv2	LVQ2 weight learning function.
learnos	Outstar weight learning function.
learnp	Perceptron weight and bias learning function.
learnpn	Normalized perceptron weight and bias learning function.
learnsom	Self-organizing map weight learning function.
learnwh	Widrow-Hoff weight and bias learning rule.

Line Search Functions	
srchbac	One-dim. minimization using backtracking search.
srchbre	One-dim. interval location using Brent's method.
srchcha	One-dim. minimization using Charalambous' method.
srchgol	One-dim. minimization using Golden section search.
srchhyb	One-dim. minimization using Hybrid bisection/cubic search.

Net Input Derivative Functions	
dnetprod	Product net input derivative function.
dnetsum	Sum net input derivative function.

Net Input Functions	
netprod	Product net input function.
netsum	Sum net input function.

Network Initialization Functions	
initlay	Layer-by-layer network initialization function.

Network Use Functions	
adapt	Allow a neural network to adapt.
disp	Display a neural network's properties.
display	Display a neural network variable's name and properties.
init	Initialize a neural network.
sim	Simulate a neural network.
train	Train a neural network.

New Networks Functions	
network	Create a custom neural network.
newc	Create a competitive layer.
newcf	Create a cascade-forward backpropagation network.
newelm	Create an Elman backpropagation network.

New Networks Functions	
newff	Create a feed-forward backpropagation network.
newfftd	Create a feed-forward input-delay backprop network.
newgrnn	Design a generalized regression neural network.
newhop	Create a Hopfield recurrent network.
newlin	Create a linear layer.
newlind	Design a linear layer.
newlvq	Create a learning vector quantization network
newp	Create a perceptron.
newpnn	Design a probabilistic neural network.
newrb	Design a radial basis network.
newrbe	Design an exact radial basis network.
newsom	Create a self-organizing map.

Performance Derivative Functions	
dmae	Mean absolute error performance derivative function.
dmse	Mean squared error performance derivatives function.
dmsereg	Mean squared error w/reg performance derivative function.
dsse	Sum squared error performance derivative function.

Performance Functions	
mae	Mean absolute error performance function.
mse	Mean squared error performance function.
msereg	Mean squared error w/reg performance function.
sse	Sum squared error performance function.

Plotting Functions	
hintonw	Hinton graph of weight matrix.
hintonwb	Hinton graph of weight matrix and bias vector.
plotbr	Plot network perf. for Bayesian regularization training.
plotep	Plot weight and bias position on error surface.
plotes	Plot error surface of single input neuron.
plotpc	Plot classification line on perceptron vector plot.
plotperf	Plot network performance.
plotpv	Plot perceptron input target vectors.
plotsom	Plot self-organizing map.
plotv	Plot vectors as lines from the origin.
plotvec	Plot vectors with different colors.

Pre and Post Processing Functions	
postmnmx	Unnormalize data which has been norm. by prenmmx.
postreg	Postprocess network response w. linear regression analysis.
poststd	Unnormalize data which has been normalized by prestd.
premnmx	Normalize data for maximum of 1 and minimum of -1.
prepca	Principal component analysis on input data.
prestd	Normalize data for unity standard deviation and zero mean.
tramnmx	Transform data with precalculated minimum and max.
trapca	Transform data with PCA matrix computed by prepca.
trastd	Transform data with precalc. mean & standard deviation.

Simulink Support Function	
gensim	Generate a Simulink block for neural network simulation.

Topology Functions	
gridtop	Gridtop layer topology function.
hextop	Hexagonal layer topology function.
randtop	Random layer topology function.

Training Functions	
trainb	Batch training with weight and bias learning rules.
trainbfg	BFGS quasi-Newton backpropagation.
trainbr	Bayesian regularization.
trainc	Cyclical order incremental update.
traincgb	Powell-Beale conjugate gradient backpropagation.
traincgf	Fletcher-Powell conjugate gradient backpropagation.
traincgp	Polak-Ribiere conjugate gradient backpropagation.
traingd	Gradient descent backpropagation.
traingda	Gradient descent with adaptive lr backpropagation.
traingdm	Gradient descent with momentum backpropagation.
traingdx	Gradient descent with momentum & adaptive lr backprop.
trainlm	Levenberg-Marquardt backpropagation.
trainoss	One step secant backpropagation.
trainr	Random order incremental update.
trainrp	Resilient backpropagation (Rprop).
trains	Sequential order incremental update.
trainscg	Scaled conjugate gradient backpropagation.

Transfer Derivative Functions	
dhardlim	Hard limit transfer derivative function.
dhardlms	Symmetric hard limit transfer derivative function.

Transfer Derivative Functions	
dlogsig	Log sigmoid transfer derivative function.
dposlin	Positive linear transfer derivative function.
dpurelin	Linear transfer derivative function.
dradbas	Radial basis transfer derivative function.
dsatlin	Saturating linear transfer derivative function.
dsatlins	Symmetric saturating linear transfer derivative function.
dtansig	Hyperbolic tangent sigmoid transfer derivative function.
dtribas	Triangular basis transfer derivative function.

Transfer Functions	
compet	Competitive transfer function.
hardlim	Hard limit transfer function.
hardlims	Symmetric hard limit transfer function.
logsig	Log sigmoid transfer function.
poslin	Positive linear transfer function.
purelin	Hard limit transfer function.
radbas	Radial basis transfer function.
satlin	Saturating linear transfer function.
satlins	Symmetric saturating linear transfer function.
softmax	Softmax transfer function.
tansig	Hyperbolic tangent sigmoid transfer function.
tribas	Triangular basis transfer function.

Utility Functions	
calca	Calculate network outputs and other signals.
calca1	Calculate network signals for one time step.
calce	Calculate layer errors.
calce1	Calculate layer errors for one time step.
calcgx	Calc. weight and bias perform. gradient as a single vector.
calcjejj	Calculate Jacobian performance vector.
calcjx	Calculate weight and bias performance Jacobian as a single matrix.
calcpd	Calculate delayed network inputs.
calcperf	Calculation network outputs, signals, and performance.
formx	Form bias and weights into single vector.
getx	Get all network weight and bias values as a single vector.
setx	Set all network weight and bias values with a single vector.

Vector Functions	
cell2mat	Combine a cell array of matrices into one matrix.
combvec	Create all combinations of vectors.
con2seq	Converts concurrent vectors to sequential vectors.
concur	Create concurrent bias vectors.
ind2vec	Convert indices to vectors.
mat2cell	Break matrix up into cell array of matrices.

Vector Functions	
minmax	Ranges of matrix rows.
normc	Normalize columns of matrix.
normr	Normalize rows of matrix.
pnormc	Pseudo-normalize columns of matrix.
quant	Discretize value as multiples of a quantity.
seq2con	Convert sequential vectors to concurrent vectors.
sumsqr	Sum squared elements of matrix.
vec2ind	Convert vectors to indices.

Weight and Bias Initialization Functions	
initcon	Conscience bias initialization function.
initzero	Zero weight and bias initialization function.
midpoint	Midpoint weight initialization function.
randnc	Normalized column weight initialization function.
randnr	Normalized row weight initialization function.
rands	Symmetric random weight/bias initialization function.
revert	Change ntwk wts. and biases to prev. initialization values.

Weight Derivative Function	
ddotprod	Dot product weight derivative function.

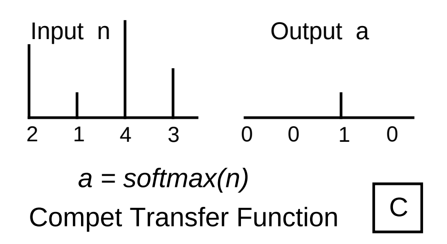
Weight Functions	
dist	Euclidean distance weight function.
dotprod	Dot product weight function.
mandist	Manhattan distance weight function.
negdist	Negative distance weight function.
normprod	Normalized dot product weight function.

Transfer Functions

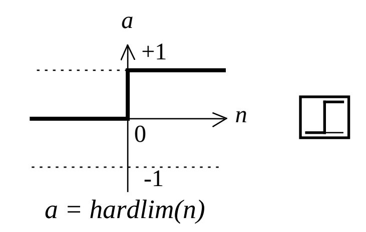
Transfer Function		
compet	Competitive transfer function.	C
hardlim	Hard limit transfer function.	
hardlims	Symmetric hard limit transfer function	
logsig	Log sigmoid transfer function.	
poslin	Positive linear transfer function	
purelin	Linear transfer function.	
radbas	Radial basis transfer function.	
satlin	Saturating linear transfer function.	

Transfer Function		
`satlins`	Symmetric saturating linear transfer function	
`softmax`	Softmax transfer function.	S
`tansig`	Hyperbolic tangent sigmoid transfer function.	
`tribas`	Triangular basis transfer function.	

Transfer Function Graphs

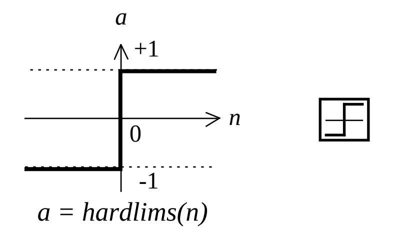


Compet Transfer Function

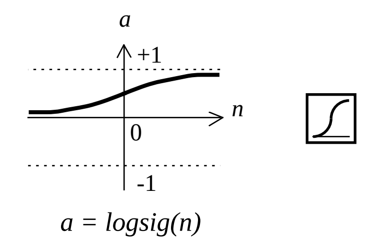


$a = hardlim(n)$

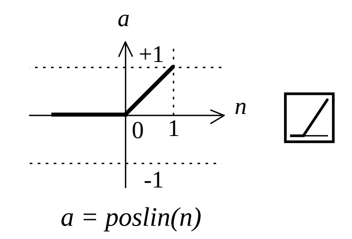
Hard-Limit Transfer Function



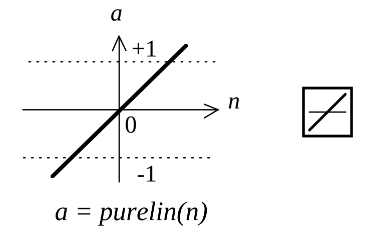
Symmetric Hard-Limit Trans. Funct.



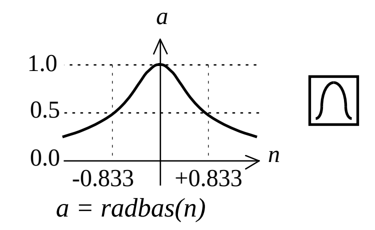
Log-Sigmoid Transfer Function



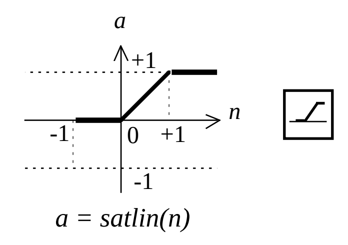
Positive Linear Transfer Funct.



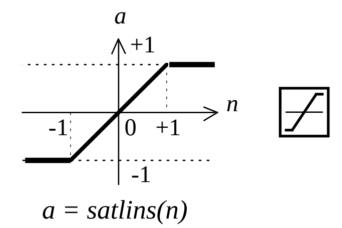
Linear Transfer Function



Radial Basis Function

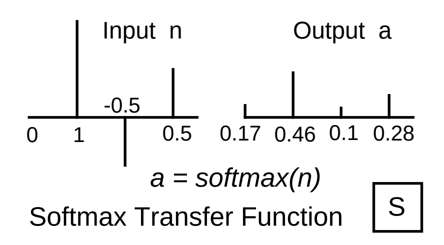


Satlin Transfer Function



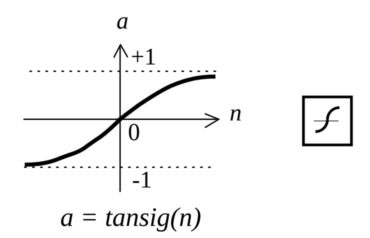
a = satlins(n)

Satlins Transfer Function



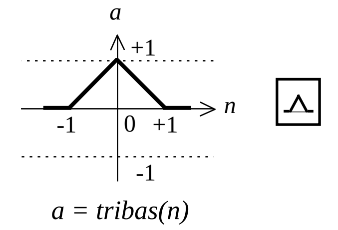
a = softmax(n)

Softmax Transfer Function



a = tansig(n)

Tan-Sigmoid Transfer Function



$a = tribas(n)$

Triangular Basis Function

Functions — Alphabetical List

The function reference pages appear in this section, listed in alphabetical order.

Reference Headings

Following is a list of headings used in the function reference pages. Not every function will have all this material, but the material that is included will be ordered as shown.

- **Purpose**
- **Graph and Symbol**
- **Syntax**
- **To Get Help**
- **Description**
- **Properties**
- **Examples**
- **Network Use**
- **Algorithm**
- **Limitations**
- **Notes**
- **See Also**
- **References**

Purpose

Allow a neural network to adapt (change weights and biases on each presentation of an input)

Syntax

```
[net,Y,E,Pf,Af] = adapt(net,P,T,Pi,Ai)
```

To Get Help

Type `help network/adapt`

Description

This function calculates network outputs and errors after each presentation of an input.

`[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)` takes,

`net`	-	Network.
`P`	-	Network inputs.
`T`	-	Network targets, default = zeros.
`Pi`	-	Initial input delay conditions, default = zeros.
`Ai`	-	Initial layer delay conditions, default = zeros.

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

`net`	-	Updated network.
`Y`	-	Network outputs.
`E`	-	Network errors.
`Pf`	-	Final input delay conditions.
`Af`	-	Final layer delay conditions.
`tr`	-	Training record (epoch and perf).

Note that `T` is optional and only needs to be used for networks that require targets. `Pi` and `Pf` are also optional and only need to be used for networks that have input or layer delays.

`adapt`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

`P  - Ni x TS` cell array, each element `P{i,ts}` is an `Ri x Q` matrix.

`T  - Nt x TS` cell array, each element `T{i,ts}` is a `Vi x Q` matrix.

`Pi - Ni x ID` cell array, each element `Pi{i,k}` is an `Ri x Q` matrix.

`Ai - Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix.

`Y  - NO x TS` cell array, each element `Y{i,ts}` is a `Ui x Q` matrix.

`E  - Nt x TS` cell array, each element `E{i,ts}` is a `Vi x Q` matrix.

`Pf - Ni x ID` cell array, each element `Pf{i,k}` is an `Ri x Q` matrix.

`Af - Nl x LD` cell array, each element `Af{i,k}` is an `Si x Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
No = net.numOutputs
Nt = net.numTargets
ID = net.numInputDelays
LD = net.numLayerDelays
```

`TS` = Number of time steps

`Q` = Batch size

```
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Ui = net.outputs{i}.size
Vi = net.targets{i}.size
```

The columns of `Pi`, `Pf`, `Ai`, and `Af` are ordered from oldest delay condition to most recent:

`Pi{i,k}` = input `i` at time `ts = k-ID`.

`Pf{i,k}` = input `i` at time `ts = TS+k-ID`.

`Ai{i,k}` = layer output `i` at time `ts = k-LD`.

`Af{i,k}` = layer output `i` at time `ts = TS+k-LD`.

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for network's with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

- P - (sum of Ri) x Q matrix
- T - (sum of Vi) x Q matrix
- Pi - (sum of Ri) x (ID*Q) matrix.
- Ai - (sum of Si) x (LD*Q) matrix.
- Y - (sum of Ui) x Q matrix.
- Pf - (sum of Ri) x (ID*Q) matrix.
- Af - (sum of Si) x (LD*Q) matrix.

Examples

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.

```
p1 = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t1 = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
```

Here newlin is used to create a layer with an input range of [-1 1]), one neuron, input delays of 0 and 1, and a learning rate of 0.5. The linear layer is then simulated.

```
net = newlin([-1 1],1,[0 1],0.5);
```

Here the network adapts for one pass through the sequence.

The network's mean squared error is displayed. (Since this is the first call of adapt, the default Pi is used.)

```
[net,y,e,pf] = adapt(net,p1,t1);
mse(e)
```

Note the errors are quite large. Here the network adapts to another 12 time steps (using the previous Pf as the new initial delay conditions.)

```
p2 = {1 -1 -1 1 1 -1  0 0 0 1 -1 -1};
t2 = {2  0 -2 0 2  0 -1 0 0 1  0 -1};
[net,y,e,pf] = adapt(net,p2,t2,pf);
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```
p3 = [p1 p2];
t3 = [t1 t2];
net.adaptParam.passes = 100;
[net,y,e] = adapt(net,p3,t3);
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

Algorithm

`adapt` calls the function indicated by `net.adaptFcn`, using the adaption parameter values indicated by `net.adaptParam`.

Given an input sequence with `TS` steps, the network is updated as follows. Each step in the sequence of inputs is presented to the network one at a time. The network's weight and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated `TS` times.

See Also

`sim`, `init`, `train`, `revert`

Purpose Box distance function

Syntax

```
d = boxdist(pos);
```

Description

boxdist is a layer distance function that is used to find the distances between the layer's neurons, given their positions.

boxdist(pos) takes one argument,

pos - N x S matrix of neuron positions.

and returns the S x S matrix of distances.

boxdist is most commonly used in conjunction with layers whose topology function is gridtop.

Examples

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
d = boxdist(pos)
```

Network Use

You can create a standard network that uses boxdist as a distance function by calling newsom.

To change a network so that a layer's topology uses boxdist, set net.layers{i}.distanceFcn to 'boxdist'.

In either case, call sim to simulate the network with boxdist. See newsom for training and adaption examples.

Algorithm

The box distance D between two position vectors Pi and Pj from a set of S vectors is:

```
Dij = max(abs(Pi-Pj))
```

See Also

sim, dist, mandist, linkdist

calca

Purpose Calculate network outputs and other signals

Syntax `[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,Q,TS)`

Description This function calculates the outputs of each layer in response to a network's delayed inputs and initial layer delay conditions.

`[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,Q,TS)` takes,

- `net` - Neural network.
- `Pd` - Delayed inputs.
- `Ai` - Initial layer delay conditions.
- `Q` - Concurrent size.
- `TS` - Time steps.

and returns,

- `Ac` - Combined layer outputs = [`Ai`, calculated layer outputs].
- `N` - Net inputs.
- `LWZ` - Weighted layer outputs.
- `IWZ` - Weighted inputs.
- `BZ` - Concurrent biases.

Examples Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],3,[0 2 4]);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here the two initial layer delay conditions for each of the three neurons are defined:

```
Ai = {[0.5; 0.1; 0.2] [0.6; 0.5; 0.2]};
```

Here we calculate the network's combined outputs Ac, and other signals described above.

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,8)
```

calca1

Purpose

Calculate network signals for one time step

Syntax

```
[Ac,N,LWZ,IWZ,BZ] = calca1(net,Pd,Ai,Q)
```

Description

This function calculates the outputs of each layer in response to a network's delayed inputs and initial layer delay conditions, for a single time step.

Calculating outputs for a single time step is useful for sequential iterative algorithms such as `trains`, which need to calculate the network response for each time step individually.

`[Ac,N,LWZ,IWZ,BZ] = calca1(net,Pd,Ai,Q)` takes,

- `net` - Neural network.
- `Pd` - Delayed inputs for a single time step.
- `Ai` - Initial layer delay conditions for a single time step.
- `Q` - Concurrent size.

and returns,

- `A` - Layer outputs for the time step.
- `N` - Net inputs for the time step.
- `LWZ` - Weighted layer outputs for the time step.
- `IWZ` - Weighted inputs for the time step.
- `BZ` - Concurrent biases for the time step.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],3,[0 2 4]);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
Pi = {0.2 0.3 0.4 0.1};
```

```
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here the two initial layer delay conditions for each of the three neurons are defined:

```
Ai = {[0.5; 0.1; 0.2] [0.6; 0.5; 0.2]};
```

Here we calculate the network's combined outputs Ac, and other signals described above.

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,8)
```

calce

Purpose

Calculate layer errors

Syntax

```
El = calce(net,Ac,Tl,TS)
```

Description

This function calculates the errors of each layer in response to layer outputs and targets.

`El = calce(net,Ac,Tl,TS)` takes,

- `net` - Neural network.
- `Ac` - Combined layer outputs.
- `Tl` - Layer targets.
- `Q` - Concurrent size.

and returns,

- `El` - Layer errors.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined, and the networks combined outputs `Ac` and other signals are calculated.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,5);
```

Here we define the layer targets for the two neurons for each of the five time steps, and calculate the layer errors.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
El = calce(net,Ac,Tl,5)
```

Here we view the network's error for layer 1 at time step 2.

```
El{1,2}
```

calce1

Purpose

Calculate layer errors for one time step

Syntax

```
El = calce1(net,A,Tl)
```

Description

This function calculates the errors of each layer in response to layer outputs and targets, for a single time step. Calculating errors for a single time step is useful for sequential iterative algorithms such as `trains` which need to calculate the network response for each time step individually.

`El = calce1(net,A,Tl)` takes,

- `net` - Neural network.
- `A` - Layer outputs, for a single time step.
- `Tl` - Layer targets, for a single time step.

and returns,

- `El` - Layer errors, for a single time step.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence P with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined, and the networks combined outputs `Ac` and other signals are calculated.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
```

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,5);
```

Here we define the layer targets for the two neurons for each of the five time steps, and calculate the layer error using the first time step layer output `Ac(:,5)` (The five is found by adding the number of layer delays, 2, to the time step 1.), and the first time step targets `Tl(:,1)`.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
El = calce1(net,Ac(:,3),Tl(:,1))
```

Here we view the network's error for layer 1.

```
El{1}
```

calcgx

Purpose Calculate weight and bias performance gradient as a single vector

Syntax

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS);
```

Description This function calculates the gradient of a network's performance with respect to its vector of weight and bias values X.

If the network has no layer delays with taps greater than 0 the result is the true gradient.

If the network as layer delays greater than 0, the result is the Elman gradient, an approximation of the true gradient.

`[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS)` takes,

- `net` - Neural network.
- `X` - Vector of weight and bias values.
- `Pd` - Delayed inputs.
- `BZ` - Concurrent biases.
- `IWZ` - Weighted inputs.
- `LWZ` - Weighted layer outputs.
- `N` - Net inputs.
- `Ac` - Combined layer outputs.
- `El` - Layer errors.
- `perf` - Network performance.
- `Q` - Concurrent size.
- `TS` - Time steps.

and returns,

- `gX` - Gradient dPerf/dX.
- `normgX` - Norm of gradient.

Examples Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
```

```
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
X = getx(net);
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcgz` to calculate the gradient of performance with respect to the weight and bias values `X`.

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,1,5);
```

See Also

`calcjx`, `calcjejj`

Purpose

Calculate Jacobian performance vector

Syntax

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)
```

Description

This function calculates two values (related to the Jacobian of a network) required to calculate the network's Hessian, in a memory efficient way.

Two values needed to calculate the Hessian of a network are J*E (Jacobian times errors) and J'J (Jacobian squared). However the Jacobian J can take up a lot of memory. This function calculates J*E and J'J by dividing up training vectors into groups, calculating partial Jacobians Ji and its associated values Ji*Ei and Ji'Ji, then summing the partial values into the full J*E and J'J values.

This allows the J*E and J'J values to be calculated with a series of smaller Ji matrices, instead of a larger J matrix.

`[je,jj,normgX] = calcjejj(net,PD,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)` takes,

- `net` - Neural network.
- `PD` - Delayed inputs.
- `BZ` - Concurrent biases.
- `IWZ` - Weighted inputs.
- `LWZ` - Weighted layer outputs.
- `N` - Net inputs.
- `Ac` - Combined layer outputs.
- `El` - Layer errors.
- `Q` - Concurrent size.
- `TS` - Time steps.
- `MR` - Memory reduction factor

and returns,

- `je` - Jacobian times errors.
- `jj` - Jacobian transposed time the `Jacobian.normgX`
- `normgX` - Norm of gradient

Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcgx` to calculate the Jacobian times error, Jacobian squared, and the norm of the Jocobian times error using a memory reduction of 2.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,2);
```

The results should be the same whatever the memory reduction used. Here a memory reduction of 3 is used.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,3);
```

See Also

`calcjx`, `calcjejj`

calcjx

Purpose

Calculate weight and bias performance Jacobian as a single matrix

Syntax

```
jx = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS)
```

Description

This function calculates the Jacobian of a network's errors with respect to its vector of weight and bias values X.

[jX] = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS) takes,

net - Neural network.
PD - Delayed inputs.
BZ - Concurrent biases.
IWZ - Weighted inputs.
LWZ - Weighted layer outputs.
N - Net inputs.
Ac - Combined layer outputs.
Q - Concurrent size.
TS - Time steps.

and returns,

jX - Jacobian of network errors with respect to X.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (Q = 1) input sequence P with five time steps (TS = 5), and the four initial input delay conditions Pi, combined inputs Pc, and delayed inputs Pd.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcjx` to calculate the Jacobian.

```
jX = calcjx(net,Pd,BZ,IWZ,LWZ,N,Ac,1,5);calcpd
```

See Also

`calcgx`, `calcjejj`

calcpd

Purpose

Calculate delayed network inputs

Syntax

```
Pd = calcpd(net,TS,Q,Pc)
```

Description

This function calculates the results of passing the network inputs through each input weights tap delay line.

`Pd = calcpd(net,TS,Q,Pc)` takes,

- `net` - Neural network.
- `TS` - Time steps.
- `Q` - Concurrent size.
- `Pc` - Combined inputs = [initial delay conditions, network inputs].

and returns,

- `Pd` - Delayed inputs.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps.

```
net = newlin([0 1],3,[0 2 4]);
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`).

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
```

Here we define the four initial input delay conditions `Pi`.

```
Pi = {0.2 0.3 0.4 0.1};
```

The delayed inputs (the inputs after passing through the tap delays) can be calculated with `calcpd`.

```
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here we view the delayed inputs for input weight going to layer 1, from input 1 at time steps 1 and 2.

```
Pd{1,1,1}
Pd{1,1,2}
```

Purpose

Calculate network outputs, signals, and performance

Syntax

```
[perf,El,Ac,N,BZ,IWZ,LWZ]=calcperf(net,X,Pd,Tl,Ai,Q,TS)
```

Description

This function calculates the outputs of each layer in response to a networks delayed inputs and initial layer delay conditions.

`[perf,El,Ac,N,LWZ,IWZ,BZ] = calcperf(net,X,Pd,Tl,Ai,Q,TS)` takes,

- `net` - Neural network.
- `X` - Network weight and bias values in a single vector.
- `Pd` - Delayed inputs.
- `Tl` - Layer targets.
- `Ai` - Initial layer delay conditions.
- `Q` - Concurrent size.
- `TS` - Time steps.

and returns,

- `perf` - Network performance.
- `El` - Layer errors.
- `Ac` - Combined layer outputs = [`Ai`, calculated layer outputs].
- `N` - Net inputs.
- `LWZ` - Weighted layer outputs.
- `IWZ` - Weighted inputs.
- `BZ` - Concurrent biases.

Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`),and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
```

Here we define the layer targets for the two neurons for each of the five time steps.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted.

```
X = getx(net);
```

Here we calculate the network's combined outputs `Ac`, and other signals described above.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5)
```

Purpose Create all combinations of vectors

Syntax

```
combvec(a1,a2...)
```

Description

`combvec(A1,A2...)` takes any number of inputs,

`A1` - Matrix of `N1` (column) vectors

`A2` - Matrix of `N2` (column) vectors

and returns a matrix of `(N1*N2*...)` column vectors, where the columns consist of all possibilities of `A2` vectors, appended to `A1` vectors, etc.

Examples

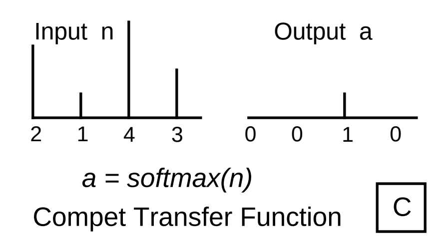
```
a1 = [1 2 3; 4 5 6];
a2 = [7 8; 9 10];
a3 = combvec(a1,a2)
```

compet

Purpose

Competitive transfer function

Graph and Symbol



Syntax

```
A = compet(N)
info = compet(code)
```

Description

`compet` is a transfer function. Transfer functions calculate a layer's output from its net input.

`compet(N)` takes one input argument,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns output vectors with 1 where each net input vector has its maximum value, and 0 elsewhere.

`compet(code)` returns information about this function.

These codes are defined:

`'deriv'` - Name of derivative function.

`'name'` - Full name.

`'output'` - Output range.

`'active'` - Active input range.

`compet` does not have a derivative function.

In many network paradigms it is useful to have a layer whose neurons compete for the ability to output a 1. In biology this is done by strong inhibitory connections between each of the neurons in a layer. The result is that the only neuron that can respond with appreciable output is the neuron whose net input is the highest. All other neurons are inhibited so strongly by the *winning* neuron that their outputs are negligible.

To model this type of layer efficiently on a computer, a competitive transfer function is often used. Such a function transforms the net input vector of a layer of neurons so that the neuron receiving the greatest net input has an output of 1 and all other neurons have outputs of 0.

Examples

Here we define a net input vector N, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = compet(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

Network Use

You can create a standard network that uses compet by calling newc or newpnn.

To change a network so a layer uses compet, set net.layers{i,j}.transferFcn to 'compet'.

In either case, call sim to simulate the network with compet.

See newc or newpnn for simulation examples.

See Also

sim, softmax

con2seq

Purpose Convert concurrent vectors to sequential vectors

Syntax

```
s = con2seq(b)
```

Description The Neural Network Toolbox arranges concurrent vectors with a matrix, and sequential vectors with a cell array (where the second index is the time step).

`con2seq` and `seq2con` allow concurrent vectors to be converted to sequential vectors, and back again.

`con2seq(b)`takes one input,

`b` - `R` x `TS` matrix.

and returns one output,

`S` - `1` x `TS` cell array of `R` x 1 vectors.

`con2seq(b,TS)` can also convert multiple batches,

`b` - `N` x `1` cell array of matrices with `M*TS` columns.

`TS` - Time steps.

and will return,

`S` - `N` x `TS` cell array of matrices with `M` columns.

Examples Here a batch of three values is converted to a sequence.

```
p1 = [1 4 2]
p2 = con2seq(p1)
```

Here two batches of vectors are converted to two sequences with two time steps.

```
p1 = {[1 3 4 5; 1 1 7 4]; [7 3 4 4; 6 9 4 1]}
p2 = con2seq(p1,2)
```

See Also `seq2con`, `concur`

concur

Purpose Create concurrent bias vectors

Syntax

```
concur(b,q)
```

Description

`concur(B,Q)`

- `B` - `S` x 1 bias vector (or `Nl x 1` cell array of vectors).
- `Q` - Concurrent size.

Returns an `S` x `B` matrix of copies of `B` (or `Nl x 1` cell array of matrices).

Examples

Here `concur` creates three copies of a bias vector.

```
b = [1; 3; 2; -1];
concur(b,3)
```

Network Use

To calculate a layer's net input, the layer's weighted inputs must be combined with its biases. The following expression calculates the net input for a layer with the `netsum` net input function, two input weights, and a bias:

```
n = netsum(z1,z2,b)
```

The above expression works if `Z1`, `Z2`, and `B` are all `S x 1` vectors. However, if the network is being simulated by `sim` (or `adapt` or `train`) in response to `Q` concurrent vectors, then `Z1` and `Z2` will be `S x Q` matrices. Before `B` can be combined with `Z1` and `Z2`, we must make `Q` copies of it.

```
n = netsum(z1,z2,concur(b,q))
```

See Also

`netsum, netprod, sim, seq2con, con2seq`

ddotprod

Purpose Dot product weight derivative function

Syntax

```
dZ_dP = ddotprod('p',W,P,Z)
dZ_dW = ddotprod('w',W,P,Z)
```

Description

`ddotprod` is a weight derivative function.

`ddotprod('p',W,P,Z)` takes three arguments,

- `W` - `S` x `R` weight matrix.
- `P` - `R` x `Q` Inputs.
- `Z` - `S` x `Q` weighted input.

and returns the `S` x `R` derivative `dZ/dP`.

`ddotprod('w',W,P,Z)` returns the `R` x `Q` derivative `dZ/dW`.

Examples

Here we define a weight `W` and input `P` for an input with three elements and a layer with two neurons.

```
W = [0 -1 0.2; -1.1 1 0];
P = [0.1; 0.6; -0.2];
```

Here we calculate the weighted input with `dotprod`, then calculate each derivative with `ddotprod`.

```
Z = dotprod(W,P)
dZ_dP = ddotprod('p',W,P,Z)
dZ_dW = ddotprod('w',W,P,Z)
```

Algorithm

The derivative of a product of two elements with respect to one element is the other element.

```
dZ/dP = W
dZ/dW = P
```

See Also

`dotprod`

dhardlim

Purpose Derivative of hard limit transfer function

Syntax

```
dA_dN = dhardlim(N,A)
```

Description

`dhardlim` is the derivative function for `hardlim`.

`dhardlim(N,A)` takes two arguments,

`N` - `S` x `Q` net input.

`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples

Here we define the net input `N` for a layer of 3 `hardlim` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `hardlim` and then the derivative of `A` with respect to `N`.

```
A = hardlim(N)
dA_dN = dhardlim(N,A)
```

Algorithm

The derivative of `hardlim` is calculated as follows:

```
d = 0
```

See Also

`hardlim`

dhardlms

Purpose Derivative of symmetric hard limit transfer function

Syntax

```
dA_dN = dhardlms(N,A)
```

Description

`dhardlms` is the derivative function for `hardlims`.

`dhardlms(N,A)` takes two arguments,

`N` - `S` x `Q` net input.

`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples

Here we define the net input `N` for a layer of 3 `hardlims` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `hardlims` and then the derivative of `A` with respect to `N`.

```
A = hardlims(N)
dA_dN = dhardlms(N,A)
```

Algorithm

The derivative of `hardlims` is calculated as follows:

```
d = 0
```

See Also

`hardlims`

disp

Purpose Display a neural network's properties

Syntax

```
disp(net)
```

To Get Help Type `help network/disp`

Description `disp(net)` displays a network's properties.

Examples Here a perceptron is created and displayed.

```
net = newp([-1 1; 0 2],3);
disp(net)
```

See Also `display, sim, init, train, adapt`

display

Purpose

Display the name and properties of a neural network's variables

Syntax

```
display(net)
```

To Get Help

Type `help network/disp`

Description

`display(net)` displays a network variable's name and properties.

Examples

Here a perceptron variable is defined and displayed.

```
net = newp([-1 1; 0 2],3);
display(net)
```

`display` is automatically called as follows:

```
net
```

See Also

`disp`, `sim`, `init`, `train`, `adapt`

Purpose

Euclidean distance weight function

Syntax

```
Z = dist(W,P)
df = dist('deriv')
D = dist(pos)
```

Description

`dist` is the Euclidean distance weight function. Weight functions apply weights to an input to get weighted inputs.

`dist (W,P)` takes these inputs,

`W` - `S` x `R` weight matrix.

`P` - `R` x `Q` matrix of `Q` input (column) vectors.

and returns the `S` x `Q` matrix of vector distances.

`dist('deriv')` returns `''` because `dist` does not have a derivative function.

`dist` is also a layer distance function, which can be used to find the distances between neurons in a layer.

`dist(pos)` takes one argument,

`pos` - `N` x `S` matrix of neuron positions.

and returns the `S` x `S` matrix of distances.

Examples

Here we define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = dist(W,P)
```

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = dist(pos)
```

Network Use

You can create a standard network that uses `dist` by calling `newpnn` or `newgrnn`.

To change a network so an input weight uses `dist`, set `net.inputWeight{i,j}.weightFcn` to '`dist`'.

For a layer weight set `net.inputWeight{i,j}.weightFcn` to '`dist`'.

To change a network so that a layer's topology uses `dist`, set `net.layers{i}.distanceFcn` to '`dist`'.

In either case, call `sim` to simulate the network with `dist`.

See `newpnn` or `newgrnn` for simulation examples.

Algorithm

The Euclidean distance d between two vectors `X` and `Y` is:

```
d = sum((x-y).^2).^0.5
```

See Also

`sim, dotprod, negdist, normprod, mandist, linkdist`

Purpose Log sigmoid transfer derivative function

Syntax

```
dA_dN = dlogsig(N,A)
```

Description `dlogsig` is the derivative function for `logsig`.

`dlogsig(N,A)` takes two arguments,

`N - S x Q` net input.

`A - S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples Here we define the net input `N` for a layer of 3 `tansig` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `logsig` and then the derivative of `A` with respect to `N`.

```
A = logsig(N)
dA_dN = dlogsig(N,A)
```

Algorithm The derivative of `logsig` is calculated as follows:

```
d = a * (1 - a)
```

See Also `logsig, tansig, dtansig`

dmae

Purpose

Mean absolute error performance derivative function

Syntax

```
dPerf_dE = dmae('e',E,X,perf,PP)
dPerf_dX = dmae('x',E,X,perf,PP)
```

Description

`dmae` is the derivative function for `mae`.

`dmae('d',E,X,PERF,PP)` takes these arguments,

`E`	-	Matrix or cell array of error vector(s).
`X`	-	Vector of all weight and bias values.
`perf`	-	Network performance (ignored).
`PP`	-	Performance parameters (ignored).

and returns the derivative `dPerf/dE`.

`dmae('x',E,X,PERF,PP)` returns the derivative `dPerf/dX`.

Examples

Here we define `E` and `X` for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network's mean absolute error performance, and derivatives of performance.

```
perf = mae(E)
dPerf_dE = dmae('e',E,X)
dPerf_dX = dmae('x',E,X)
```

Note that `mae` can be called with only one argument and `dmae` with only three arguments because the other arguments are ignored. The other arguments exist so that `mae` and `dmae` conform to standard performance function argument lists.

See Also

`mae`

Purpose

Mean squared error performance derivatives function

Syntax

```
dPerf_dE = dmse('e',E,X,perf,PP)
dPerf_dX = dmse('x',E,X,perf,PP)
```

Description

`dmse` is the derivative function for `mse`.

`dmse('d',E,X,PERF,PP)` takes these arguments,

`E`	-	Matrix or cell array of error vector(s).
`X`	-	Vector of all weight and bias values.
`perf`	-	Network performance (ignored).
`PP`	-	Performance parameters (ignored).

and returns the derivative `dPerf/dE`.

`dmse('x',E,X,PERF,PP)` returns the derivative `dPerf/dX`.

Examples

Here we define `E` and `X` for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network's mean squared error performance, and derivatives of performance.

```
perf = mse(E)
dPerf_dE = dmse('e',E,X)
dPerf_dX = dmse('x',E,X)
```

Note that `mse` can be called with only one argument and `dmse` with only three arguments because the other arguments are ignored. The other arguments exist so that `mse` and `dmse` conform to standard performance function argument lists.

See Also

`mse`

dmsereg

Purpose

Mean squared error with regularization or performance derivative function

Syntax

```
dPerf_dE = dmsereg('e',E,X,perf,PP)
dPerf_dX = dmsereg('x',E,X,perf,PP)
```

Description

`dmsereg` is the derivative function for `msereg`.

`dmsereg('d',E,X,perf,PP)` takes these arguments,

- `E` - Matrix or cell array of error vector(s).
- `X` - Vector of all weight and bias values.
- `perf` - Network performance (ignored).
- `PP` - `mse` performance parameter.

where `PP` defines one performance parameters,

- `PP.ratio` - Relative importance of errors vs. weight and bias values.

and returns the derivative `dPerf/dE`.

`dmsereg('x',E,X,perf)` returns the derivative `dPerf/dX`.

`mse` has only one performance parameter.

Examples

Here we define an error `E` and `X` for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here the ratio performance parameter is defined so that squared errors are 5 times as important as squared weight and bias values.

```
pp.ratio = 5/(5+1);
```

Here we calculate the network's performance, and derivatives of performance.

```
perf = msereg(E,X,pp)
dPerf_dE = dmsereg('e',E,X,perf,pp)
dPerf_dX = dmsereg('x',E,X,perf,pp)
```

See Also

`msereg`

Purpose

Derivative of net input product function

Syntax

```
dN_dZ = dnetprod(Z,N)
```

Description

`dnetprod` is the net input derivative function for `netprod`.

`dnetprod` takes two arguments,

- `Z` - `S` x `Q` weighted input.
- `N` - `S` x `Q` net input.

and returns the `S` x `Q` derivative `dN/dZ`.

Examples

Here we define two weighted inputs for a layer with three neurons.

```
Z1 = [0; 1; -1];
Z2 = [1; 0.5; 1.2];
```

We calculate the layer's net input `N` with `netprod` and then the derivative of `N` with respect to each weighted input.

```
N = netprod(Z1,Z2)
dN_dZ1 = dnetprod(Z1,N)
dN_dZ2 = dnetprod(Z2,N)
```

Algorithm

The derivative of a product with respect to any element of that product is the product of the other elements.

See Also

`netsum`, `netprod`, `dnetsum`

dnetsum

Purpose Sum net input derivative function

Syntax

```
dN_dZ = dnetsum(Z,N)
```

Description

`dnetsum` is the net input derivative function for `netsum`.

`dnetsum` takes two arguments,

`Z` - `S` x `Q` weighted input.

`N` - `S` x `Q` net input.

and returns the `S` x `Q` derivative `dN/dZ`.

Examples

Here we define two weighted inputs for a layer with three neurons.

```
Z1 = [0; 1; -1];
Z2 = [1; 0.5; 1.2];
```

We calculate the layer's net input `N` with `netsum` and then the derivative of `N` with respect to each weighted input.

```
N = netsum(Z1,Z2)
dN_dZ1 = dnetsum(Z1,N)
dN_dZ2 = dnetsum(Z2,N)
```

Algorithm

The derivative of a sum with respect to any element of that sum is always a ones matrix that is the same size as the sum.

See Also

`netsum`, `netprod`, `dnetprod`

Purpose

Dot product weight function

Syntax

```
Z = dotprod(W,P)
df = dotprod('deriv')
```

Description

dotprod is the dot product weight function. Weight functions apply weights to an input to get weighted inputs.

dotprod(W,P) takes these inputs,

W - S x R weight matrix.

P - R x Q matrix of Q input (column) vectors.

and returns the S x Q dot product of W and P.

Examples

Here we define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = dotprod(W,P)
```

Network Use

You can create a standard network that uses dotprod by calling newp or newlin.

To change a network so an input weight uses dotprod, set net.inputWeight{i,j}.weightFcn to 'dotprod'. For a layer weight, set net.inputWeight{i,j}.weightFcn to 'dotprod'.

In either case, call sim to simulate the network with dotprod.

See newp and newlin for simulation examples.

See Also

sim, ddotprod, dist, negdist, normprod

dposlin

Purpose Derivative of positive linear transfer function

Syntax

```
dA_dN = dposlin(N,A)
```

Description `dposlin` is the derivative function for `poslin`.

`dposlin(N,A)` takes two arguments, and returns the `S` x `Q` derivative `dA/dN`.

Examples Here we define the net input `N` for a layer of 3 `poslin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `poslin` and then the derivative of `A` with respect to `N`.

```
A = poslin(N)
dA_dN = dposlin(N,A)
```

Algorithm The derivative of `poslin` is calculated as follows:

d = 1, if 0 <= n; 0, Otherwise.

See Also `poslin`

dpurelin

Purpose
Linear transfer derivative function

Syntax

```
dA_dN = dpurelin(N,A)
```

Description

`dpurelin` is the derivative function for `logsig`.

`dpurelin(N,A)` takes two arguments,

- `N` - `S` x `Q` net input.
- `A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA_dN`.

Examples

Here we define the net input `N` for a layer of 3 `purelin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `purelin` and then the derivative of `A` with respect to `N`.

```
A = purelin(N)
dA_dN = dpurelin(N,A)
```

Algorithm

The derivative of `purelin` is calculated as follows:

```
D(i,q) = 1
```

See Also

`purelin`

dradbas

Purpose Derivative of radial basis transfer function

Syntax

```
dA_dN = dradbas(N,A)
```

Description

`dradbas` is the derivative function for `radbas`.

`dradbas(N,A)` takes two arguments,

`N` - `S` x `Q` net input.
`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples

Here we define the net input `N` for a layer of 3 `radbas` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `radbas` and then the derivative of `A` with respect to `N`.

```
A = radbas(N)
```

Algorithm

The derivative of `radbas` is calculated as follows:

```
d = -2*n*a
```

See Also

`radbas`

Purpose Derivative of saturating linear transfer function

Syntax

```
dA_dN = dsatlin(N,A)
```

Description `dsatlin` is the derivative function for `satlin`.

`dsatlin(N,A)` takes two arguments,

`N - S` x `Q` net input.

`A - S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples Here we define the net input `N` for a layer of 3 `satlin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `satlin` and then the derivative of `A` with respect to `N`.

```
A = satlin(N)
dA_dN = dsatlin(N,A)
```

Algorithm The derivative of `satlin` is calculated as follows:

d = 1, if 0 <= n <= 1; 0, otherwise.

See Also `satlin`

dsatlins

Purpose Derivative of symmetric saturating linear transfer function

Syntax

```
dA_dN = dsatlins(N,A)
```

Description `dsatlins` is the derivative function for `satlins`.

`dsatlins(N,A)` takes two arguments,

`N` - `S` x `Q` net input.

`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples Here we define the net input `N` for a layer of 3 `satlins` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `satlins` and then the derivative of `A` with respect to `N`.

```
A = satlins(N)
dA_dN = dsatlins(N,A)
```

Algorithm The derivative of `satlins` is calculated as follows:

d = 1, if -1 <= n <= 1; 0, otherwise.

See Also `satlins`

Purpose Sum squared error performance derivative function

Syntax

```
dPerf_dE = dsse('e',E,X,perf,PP)
dPerf_dX = dsse('x',E,X,perf,PP)
```

Description dsse is the derivative function for sse.

dsse('d',E,X,perf,PP) takes these arguments,

- E - Matrix or cell array of error vector(s).
- X - Vector of all weight and bias values.
- perf - Network performance (ignored).
- PP - Performance parameters (ignored).

and returns the derivative dPerf_dE.

dsse('x',E,X,perf,PP) returns the derivative dPerf_dX.

Examples Here we define an error E and X for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network's sum squared error performance, and derivatives of performance.

```
perf = sse(E)
dPerf_dE = dsse('e',E,X)
dPerf_dX = dsse('x',E,X)
```

Note that sse can be called with only one argument and dsse with only three arguments because the other arguments are ignored. The other arguments exist so that sse and dsse conform to standard performance function argument lists.

See Also sse

dtansig

Purpose Hyperbolic tangent sigmoid transfer derivative function

Syntax

```
dA_dN = dtansig(N,A)
```

Description

`dtansig` is the derivative function for `tansig`.

`dtansig(N,A)` takes two arguments,

`N` - `S` x `Q` net input.

`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples

Here we define the net input `N` for a layer of 3 `tansig` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `tansig` and then the derivative of `A` with respect to `N`.

```
A = tansig(N)
dA_dN = dtansig(N,A)
```

Algorithm

The derivative of `tansig` is calculated as follows:

```
d = 1-a^2
```

See Also

`tansig`, `logsig`, `dlogsig`

Purpose Derivative of triangular basis transfer function

Syntax

```
dA_dN = dtribas(N,A)
```

Description

`dtribas` is the derivative function for `tribas`.

`dtribas(N,A)` takes two arguments,

`N` - `S` x `Q` net input.

`A` - `S` x `Q` output.

and returns the `S` x `Q` derivative `dA/dN`.

Examples

Here we define the net input `N` for a layer of 3 `tribas` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `tribas` and then the derivative of `A` with respect to `N`.

```
A = tribas(N)
dA_dN = dtribas(N,A)
```

Algorithm

The derivative of `tribas` is calculated as follows:

d = 1, if -1 <= n < 0; -1, if 0 < n <= 1; 0, otherwise.

See Also

`tribas`

errsurf

Purpose Error surface of single input neuron

Syntax `errsurf(P,T,WV,BV,F)`

Description `errsurf(P,T,WV,BV,F)` takes these arguments,

- `P` - 1 x `Q` matrix of input vectors.
- `T` - 1 x `Q` matrix of target vectors.
- `WV` - Row vector of values of `W`.
- `BV` - Row vector of values of `B`.
- `F` - Transfer function (string).

and returns a matrix of error values over `WV` and `BV`.

Examples

```
p = [-6.0 -6.1 -4.1 -4.0 +4.0 +4.1 +6.0 +6.1];
t = [+0.0 +0.0 +.97 +.99 +.01 +.03 +1.0 +1.0];
wv = -1:.1:1; bv = -2.5:.25:2.5;
es = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```

See Also `plotes`

Purpose Form bias and weights into single vector

Syntax

```
X = formx(net,B,IW,LW)
```

Description This function takes weight matrices and bias vectors for a network and reshapes them into a single vector.

`X = formx(net,B,IW,LW)` takes these arguments,

- `net` - Neural network.
- `B` - Nlx1 cell array of bias vectors.
- `IW` - NlxNi cell array of input weight matrices.
- `LW` - NlxNl cell array of layer weight matrices.

and returns,

- `X` - Vector of weight and bias values

Examples Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

We can get view its weight matrices and bias vectors as follows:

```
b = net.b
iw = net.iw
lw = net.lw
```

We can put these values into a single vector as follows:

```
x = formx(net,net.b,net.iw,net.lw))
```

See Also `getx, setx`

gensim

Purpose Generate a Simulink block for neural network simulation

Syntax

```
gensim(net,st)
```

To Get Help Type `help network/gensim`

Description

`gensim(net,st)` creates a Simulink system containing a block that simulates neural network `net`.

`gensim(net,st)` takes these inputs,

- `net` - Neural network.
- `st` - Sample time (default = 1).

and creates a Simulink system containing a block that simulates neural network `net` with a sampling time of `st`.

If `net` has no input or layer delays (`net.numInputDelays` and `net.numLayerDelays` are both 0) then you can use -1 for `st` to get a continuously sampling network.

Examples

```
net = newff([0 1],[5 1]);
gensim(net)
```

getx

Purpose Get all network weight and bias values as a single vector

Syntax

```
X = getx(net)
```

Description This function gets a network's weight and biases as a vector of values.

```
X = getx(NET)
```

`NET` - Neural network.

`X` - Vector of weight and bias values.

Examples Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

We can get its weight and bias values as follows:

```
net.iw{1,1}
net.b{1}
```

We can get these values as a single vector as follows:

```
x = getx(net);
```

See Also `setx`, `formx`

gridtop

Purpose Grid layer topology function

Syntax

```
pos = gridtop(dim1,dim2,...,dimN)
```

Description

`gridtop` calculates neuron positions for layers whose neurons are arranged in an `N` dimensional grid.

`gridtop(dim1,dim2,...,dimN)` takes `N` arguments,

`dimi` - Length of layer in dimension `i`.

and returns an `N` x `S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in a 8-by-5 grid.

```
pos = gridtop(8,5); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so the layer is very disorganized as is evident in the plot generated by the following code.

```
W = rands(40,2); plotsom(W,dist(pos))
```

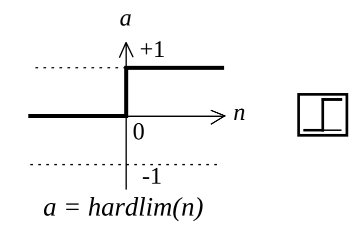
See Also

`hextop`, `randtop`

Purpose

Hard limit transfer function

Graph and Symbol



Hard-Limit Transfer Function

Syntax

```
A = hardlim(N)
info = hardlim(code)
```

Description

The hard limit transfer function forces a neuron to output a 1 if its net input reaches a threshold, otherwise it outputs 0. This allows a neuron to make a decision or classification. It can say *yes* or *no*. This kind of neuron is often trained with the perceptron learning rule.

`hardlim` is a transfer function. Transfer functions calculate a layer's output from its net input.

`hardlim(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns 1 where `N` is positive, 0 elsewhere.

`hardlim(code)` returns useful information for each `code` string,

`'deriv'` - Name of derivative function.

`'name'` - Full name.

`'output'` - Output range.

`'active'` - Active input range.

Examples

Here is the code to create a plot of the `hardlim` transfer function.

```
n = -5:0.1:5;
a = hardlim(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `hardlim` by calling `newp`.

To change a network so that a layer uses `hardlim`, set `net.layers{i}.transferFcn` to '`hardlim`'.

In either case call `sim` to simulate the network with `hardlim`.

See `newp` for simulation examples.

Algorithm

The transfer function output is one is `n` is less than or equal to 0 and zero if `n` is less than 0.

`hardlim(n)` = 1, if `n` >= 0; 0 otherwise.

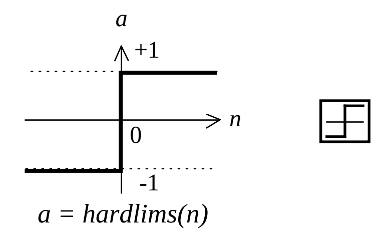
See Also

`sim, hardlims`

Purpose

Symmetric hard limit transfer function

Graph and Symbol



Symmetric Hard-Limit Trans. Funct.

Syntax

```
A = hardlims(N)
info = hardlims(code)
```

Description

The symmetric hard limit transfer function forces a neuron to output a 1 if its net input reaches a threshold. Otherwise it outputs -1. Like the regular hard limit function, this allows a neuron to make a decision or classification. It can say *yes* or *no*.

`hardlims` is a transfer function. Transfer functions calculate a layer's output from its net input.

`hardlims(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns 1 where `N` is positive, -1 elsewhere.

`hardlims(code)` return useful information for each `code` string:

`'deriv'` - Name of derivative function.

`'name'` - Full name.

`'output'` - Output range.

`'active'` - Active input range.

Examples

Here is the code to create a plot of the `hardlims` transfer function.

```
n = -5:0.1:5;
a = hardlims(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `hardlims` by calling `newp`.

To change a network so that a layer uses `hardlims`, set `net.layers{i}.transferFcn` to `'hardlims'`.

In either case call `sim` to simulate the network with `hardlims`.

See `newp` for simulation examples.

Algorithm

The transfer function output is one is `n` is greater than or equal to 0 and -1 otherwise.

`hardlim(n)` = 1, if `n` >= 0; -1 otherwise.

See Also

`sim, hardlim`

Purpose Hexagonal layer topology function

Syntax

```
pos = hextop(dim1,dim2,...,dimN)
```

Description

`hextop` calculates the neuron positions for layers whose neurons are arranged in a `N` dimensional hexagonal pattern.

`hextop(dim1,dim2,...,dimN)` takes `N` arguments,

`dimi` - Length of layer in dimension `i`.

and returns an `N`-by-`S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in a 8-by-5 hexagonal pattern.

```
pos = hextop(8,5); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so that the layer is very disorganized, as is evident in the fplo generated by the following code.

```
W = rands(40,2); plotsom(W,dist(pos))
```

See Also `gridtop`, `randtop`

hintonw

Purpose

Hinton graph of weight matrix

Syntax

```
hintonw(W,maxw,minw)
```

Description

`hintonw(W,maxw,minw)` takes these inputs,

- `W` - `S` x `R` weight matrix
- `maxw` - Maximum weight, default = `max(max(abs(W)))`.
- `minw` - Minimum weight, default = `M1/100`.

and displays a weight matrix represented as a grid of squares.

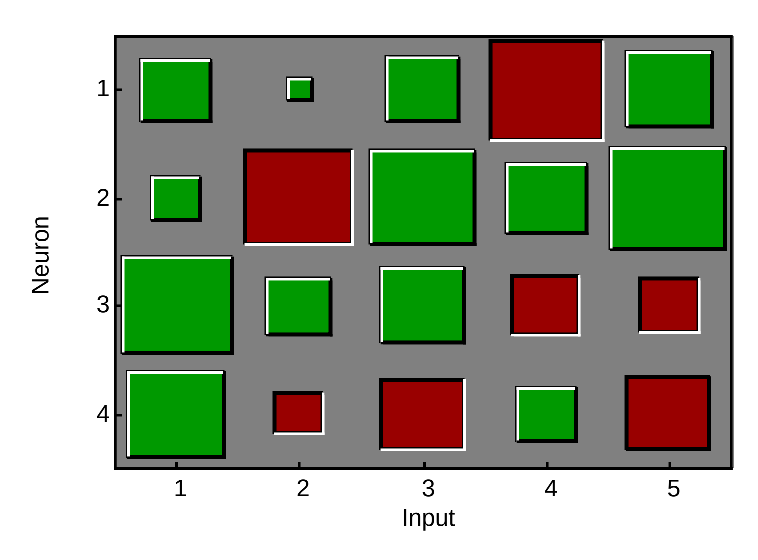
Each square's `area` represents a weight's magnitude. Each square's projection (`color`) represents a weight's sign; inset (`red`) for negative weights, projecting (`green`) for positive.

Examples

```
W = rands(4,5);
```

The following code displays the matrix graphically.

```
hintonw(W)
```



See Also

`hintonwb`

Purpose

Hinton graph of weight matrix and bias vector

Syntax

```
hintonwb(W,b,maxw,minw)
```

Description

`hintonwb(W,B,maxw,minw)` takes these inputs,

`W` - `S` x `R` weight matrix.

`B` - `S` x `1` bias vector.

`maxw` - Maximum weight, default = `max(max(abs(W)))`.

`minw` - Minimum weight, default = `M1/100`.

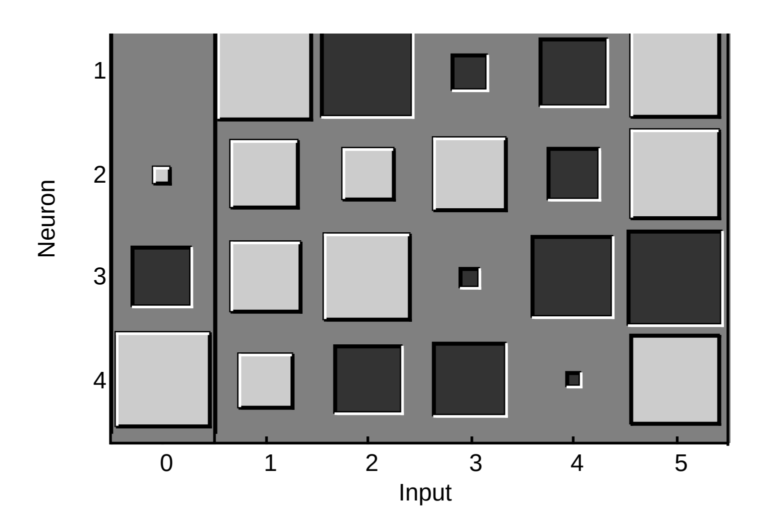
and displays a weight matrix and a bias vector represented as a grid of squares.

Each square's `area` represents a weight's magnitude. Each square's projection (`color`) represents a weight's sign; inset (`red`) for negative weights, projecting (`green`) for positive. The weights are shown on the left.

Examples

The following code produces the result shown below.

```
W = rands(4,5);
b = rands(4,1);
hintonwb(W,b)
```



See Also

`hintonw`

ind2vec

Purpose Convert indices to vectors

Syntax

```
vec = ind2vec(ind)
```

Description

`ind2vec` and `vec2ind` allow indices to either be represented by themselves, or as vectors containing a 1 in the row of the index they represent.

`ind2vec(ind)` takes one argument,

`ind` - Row vector of indices.

and returns a sparse matrix of vectors, with one 1 in each column, as indicated by `ind`.

Examples

Here four indices are defined and converted to vector representation.

```
ind = [1 3 2 3]
vec = ind2vec(ind)
```

See Also

`vec2ind`

init

Purpose Initialize a neural network

Syntax

```
net = init(net)
```

To Get Help Type `help network/init`

Description

`init(net)` returns neural network `net` with weight and bias values updated according to the network initialization function, indicated by `net.initFcn`, and the parameter values, indicated by `net.initParam`.

Examples

Here a perceptron is created with a two-element input (with ranges of 0 to 1, and -2 to 2) and 1 neuron. Once it is created we can display the neuron's weights and bias.

```
net = newp([0 1;-2 2],1);
net.iw{1,1}
net.b{1}
```

Training the perceptron alters its weight and bias values.

```
P = [0 1 0 1; 0 0 1 1];
T = [0 0 0 1];
net = train(net,P,T);
net.iw{1,1}
net.b{1}
```

`init` reinitializes those weight and bias values.

```
net = init(net);
net.iw{1,1}
net.b{1}
```

The weights and biases are zeros again, which are the initial values used by perceptron networks (see `newp`).

Algorithm

`init` calls `net.initFcn` to initialize the weight and bias values according to the parameter values `net.initParam`.

Typically, `net.initFcn` is set to `'initlay'` which initializes each layer's weights and biases according to its `net.layers{i}.initFcn`.

Backpropagation networks have `net.layers{i}.initFcn` set to '`initnw`', which calculates the weight and bias values for layer `i` using the Nguyen-Widrow initialization method.

Other networks have `net.layers{i}.initFcn` set to `'initwb'`, which initializes each weight and bias with its own initialization function. The most common weight and bias initialization function is `rands`, which generates random values between -1 and 1.

See Also `sim, adapt, train, initlay, initnw, initwb, rands, revert`

Purpose

Conscience bias initialization function

Syntax

```
b = initcon(s,pr)
```

Description

`initcon` is a bias initialization function that initializes biases for learning with the `learncon` learning function.

`initcon (S,PR)` takes two arguments,

`S` - Number of rows (neurons).

`PR` - `R` x 2 matrix of `R = [Pmin Pmax]`, default = `[1 1]`.

and returns an `S x 1` bias vector.

Note that for biases, `R` is always 1. `initcon` could also be used to initialize weights, but it is not recommended for that purpose.

Examples

Here initial bias values are calculated for a 5 neuron layer.

```
b = initcon(5)
```

Network Use

You can create a standard network that uses `initcon` to initialize weights by calling `newc`.

To prepare the bias of layer `i` of a custom network to initialize with `initcon`:

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set `net.biases{i}.initFcn` to '`initcon`'.

To initialize the network, call `init`. See `newc` for initialization examples.

Algorithm

`learncon` updates biases so that each bias value `b(i)` is a function of the average output `c(i)` of the neuron `i` associated with the bias.

`initcon` gets initial bias values by assuming that each neuron has responded to equal numbers of vectors in the "past."

See Also

`initwb, initlay, init, learncon`

initlay

Purpose Layer-by-layer network initialization function

Syntax

```
net = initlay(net)
info = initlay(code)
```

Description `initlay` is a network initialization function that initializes each layer `i` according to its own initialization function `net.layers{i}.initFcn`.

`initlay(net)` takes,

`net` - Neural network.

and returns the network with each layer updated. `initlay(code)` returns useful information for each `code` string:

`'pnames'` - Names of initialization parameters.

`'pdefaults'` - Default initialization parameters.

`initlay` does not have any initialization parameters.

Network Use You can create a standard network that uses `initlay` by calling `newp`, `newlin`, `newff`, `newcf`, and many other `new` network functions.

To prepare a custom network to be initialized with `initlay`:

1. Set `net.initFcn` to `'initlay'`. (This will set `net.initParam` to the empty matrix `[]` since `initlay` has no initialization parameters.)
2. Set each `net.layers{i}.initFcn` to a layer initialization function. (Examples of such functions are `initwb` and `initnw`).

To initialize the network, call `init`. See `newp` and `newlin` for initialization examples.

Algorithm The weights and biases of each layer `i` are initialized according to `net.layers{i}.initFcn`.

See Also `initwb`, `initnw`, `init`

Purpose Nguyen-Widrow layer initialization function

Syntax

```
net = initnw(net,i)
```

Description

`initnw` is a layer initialization function that initializes a layer's weights and biases according to the Nguyen-Widrow initialization algorithm. This algorithm chooses values in order to distribute the active region of each neuron in the layer approximately evenly across the layer's input space.

`initnw(net,i)` takes two arguments,

`net` - Neural network.

`i` - Index of a layer.

and returns the network with layer `i`'s weights and biases updated.

Network Use

You can create a standard network that uses `initnw` by calling `newff` or `newcf`.

To prepare a custom network to be initialized with `initnw`:

1. Set `net.initFcn` to '`initlay`'. (This will set `net.initParam` to the empty matrix [ ] since `initlay` has no initialization parameters.)
2. Set `net.layers{i}.initFcn` to '`initnw`'.

To initialize the network call `init`. See `newff` and `newcf` for training examples.

Algorithm

The Nguyen-Widrow method generates initial weight and bias values for a layer, so that the active regions of the layer's neurons will be distributed approximately evenly over the input space.

Advantages over purely random weights and biases are:

- Few neurons are wasted (since all the neurons are in the input space).
- Training works faster (since each area of the input space has neurons). The Nguyen-Widrow method can only be applied to layers
  - with a bias
  - with weights whose "`weightFcn`" is `dotprod`
  - with "`netInputFcn`" set to `netsum`

If these conditions are not met, then `initnw` uses `rands` to initialize the layer's weights and biases.

See Also

`initwb`, `initlay`, `init`

Purpose

By-weight-and-bias layer initialization function

Syntax

```
net = initwb(net,i)
```

Description

`initwb` is a layer initialization function that initializes a layer's weights and biases according to their own initialization functions.

`initwb(net,i)` takes two arguments,

- `net` - Neural network.
- `i` - Index of a layer.

and returns the network with layer `i`'s weights and biases updated.

Network Use

You can create a standard network that uses `initwb` by calling `newp` or `newlin`.

To prepare a custom network to be initialized with `initwb`:

1. Set `net.initFcn` to '`initlay`'. (This will set `net.initParam` to the empty matrix [ ] since `initlay` has no initialization parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set each `net.inputWeights{i,j}.initFcn` to a weight initialization function. Set each `net.layerWeights{i,j}.initFcn` to a weight initialization function. Set each `net.biases{i}.initFcn` to a bias initialization function. (Examples of such functions are `rands` and `midpoint`.)

To initialize the network, call `init`.

See `newp` and `newlin` for training examples.

Algorithm

Each weight (bias) in layer `i` is set to new values calculated according to its weight (bias) initialization function.

See Also

`initnw`, `initlay`, `init`

initzero

Purpose

Zero weight and bias initialization function

Syntax

```
W = initzero(S,PR)
b = initzero(S,[1 1])
```

Description

`initzero(S,PR)` takes two arguments,

`S` - Number of rows (neurons).

`PR` - `R` x 2 matrix of input value ranges = `[Pmin Pmax]`.

and returns an `S` x `R` weight matrix of zeros.

`initzero(S,[1 1])` returns `S` x 1 bias vector of zeros.

Examples

Here initial weights and biases are calculated for a layer with two inputs ranging over [0 1] and [-2 2], and 4 neurons.

```
W = initzero(5,[0 1; -2 2])
b = initzero(5,[1 1])
```

Network Use

You can create a standard network that uses `initzero` to initialize its weights by calling `newp` or `newlin`.

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `midpoint`:

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set each `net.inputWeights{i,j}.initFcn` to '`initzero`'. Set each `net.layerWeights{i,j}.initFcn` to '`initzero`'. Set each `net.biases{i}.initFcn` to '`initzero`'.

To initialize the network, call `init`.

See `newp` or `newlin` for initialization examples.

See Also

`initwb`, `initlay`, `init`

Purpose Conscience bias learning function

Syntax

```
[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learncon(code)
```

Description

`learncon` is the conscience bias learning function used to increase the net input to neurons that have the lowest average output until each neuron responds approximately an equal percentage of the time.

`learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `B` - `S` x `1` bias vector.
- `P` - `1`x `Q` ones vector.
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns

- `dB` - `S` x `1` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learncon`'s learning parameter, shown here with its default value.

- `LP.lr - 0.001 -` Learning rate.

`learncon(code)` returns useful information for each `code` string.

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW or gA`.

Neural Network Toolbox 2.0 compatibility: The `LP.lr` described above equals 1 minus the bias time constant used by `trainc` in Neural Network Toolbox 2.0.

Examples

Here we define a random output A, and bias vector W for a layer with 3 neurons. We also define the learning rate LR.

```
a = rand(3,1);
b = rand(3,1);
lp.lr = 0.5;
```

Since learncon only needs these values to calculate a bias change (see algorithm below), we will use them to do so.

```
dW = learncon(b,[],[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the bias of layer i of a custom network to learn with learncon:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set net.inputWeights{i}.learnFcn to 'learncon'. Set each net.layerWeights{i,j}.learnFcn to 'learncon'. (Each weight learning parameter property will automatically be set to learncon's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

Algorithm

learncon calculates the bias change db for a given neuron by first updating each neuron's *conscience*, i.e. the running average of its output:

```
c = (1-lr)*c + lr*a
```

The conscience is then used to compute a bias for the neuron that is greatest for smaller conscience values.

```
b = exp(1-log(c)) - b
```

(Note that learncon is able to recover C each time it is called from the bias values.)

See Also

learnk, learnos, adapt, train

Purpose Gradient descent weight and bias learning function

Syntax

```
[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngd(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngd(code)
```

Description

`learngd` is the gradient descent weight and bias learning function.

`learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or ones(`1`,`Q`)).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learngd`'s learning parameter shown here with its default value.

- `LP.lr - 0.01` - Learning rate.

`learngd(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random gradient gW for a weight going to a layer with 3 neurons, from an input with 2 elements. We also define a learning rate of 0.5.

```
gW = rand(3,2);
lp.lr = 0.5;
```

Since learngd only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learngd([],[],[],[],[],[],[],gW,[],[],lp,[])
```

Network Use

You can create a standard network that uses learngd with newff, newcf, or newelm. To prepare the weights and the bias of layer i of a custom network to adapt with learngd:

1. Set net.adaptFcn to 'trains'. net.adaptParam will automatically become trains's default parameters.
2. Set each net.inputWeights{i,j}.learnFcn to 'learngd'. Set each net.layerWeights{i,j}.learnFcn to 'learngd'. Set net.biases{i}.learnFcn to 'learngd'. Each weight and bias learning parameter property will automatically be set to learngd's default parameters.

To allow the network to adapt:

1. Set net.adaptParam properties to desired values.
2. Call adapt with the network.

See newff or newcf for examples.

Algorithm

learngd calculates the weight change dW for a given neuron from the neuron's input P and error E, and the weight (or bias) learning rate LR, according to the gradient descent: dw = lr*gW.

See Also

learngdm, newff, newcf, adapt, train

Purpose Gradient descent with momentum weight and bias learning function

Syntax

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngdm(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngdm(code)
```

Description `learngdm` is the gradient descent with momentum weight and bias learning function.

`learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learngdm`'s learning parameters, shown here with their default values.

- `LP.lr - 0.01` - Learning rate.
- `LP.mc - 0.9` - Momentum constant.

`learngdm(code)` returns useful information for each `code` string:

'`pnames`' - Names of learning parameters.

'`pdefaults`' - Default learning parameters.

'`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random gradient `G` for a weight going to a layer with 3 neurons, from an input with 2 elements. We also define a learning rate of 0.5 and momentum constant of 0.8;

```
gW = rand(3,2);
lp.lr = 0.5;
lp.mc = 0.8;
```

Since `learngdm` only needs these values to calculate a weight change (see algorithm below), we will use them to do so. We will use the default initial learning state.

```
ls = [];
[dW,ls] = learngdm([],[],[],[],[],[],[],gW,[],[],lp,ls)
```

`learngdm` returns the weight change and a new learning state.

Network Use

You can create a standard network that uses `learngdm` with `newff`, `newcf`, or `newelm`.

To prepare the weights and the bias of layer `i` of a custom network to adapt with `learngdm`:

1. Set `net.adaptFcn` to '`trains`'. `net.adaptParam` will automatically become `trains`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to '`learngdm`'. Set each `net.layerWeights{i,j}.learnFcn` to '`learngdm`'. Set `net.biases{i}.learnFcn` to '`learngdm`'. Each weight and bias learning parameter property will automatically be set to `learngdm`'s default parameters.

To allow the network to adapt:

1. Set `net.adaptParam` properties to desired values.
2. Call `adapt` with the network.

See `newff` or `newcf` for examples.

Algorithm

`learngdm` calculates the weight change `dW` for a given neuron from the neuron's input `P` and error `E`, the weight (or bias) `W`, learning rate `LR`, and momentum constant `MC`, according to gradient descent with momentum:

```
dW = mc*dWprev + (1-mc)*lr*gW
```

The previous weight change `dWprev` is stored and read from the learning state `LS`.

See Also

`learngd`, `newff`, `newcf`, `adapt`, `train`

learnh

Purpose Hebb weight learning rule

Syntax

```
[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnh(code)
```

Description

`learnh` is the Hebb weight learning function.

`learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S x 1` bias vector).
- `P` - `R` x `Q` input vectors (or ones(`1,Q`)).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnh`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01 -` Learning rate.

`learnh(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P and output A for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
lp.lr = 0.5;
```

Since learnh only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnh([],p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnh:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnh'. Set each net.layerWeights{i,j}.learnFcn to 'learnh'. Each weight learning parameter property will automatically be set to learnh's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

Algorithm

learnh calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the Hebb learning rule:

```
dw =  lr*a*p'
```

See Also

learnhd, adapt, train

References

Hebb, D.O., *The Organization of Behavior,* New York: Wiley, 1949.

learnhd

Purpose

Hebb with decay weight learning rule

Syntax

```
[dW,LS] = learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnhd(code)
```

Description

`learnhd` is the Hebb weight learning function.

`learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnhd`'s learning parameters shown here with default values.

- `LP.dr - 0.01` - Decay rate.
- `LP.lr - 0.1` - Learning rate.

`learnhd(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P, output A, and weights W for a layer with a two-element input and three neurons. We also define the decay and learning rates.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.dr = 0.05;
lp.lr = 0.5;
```

Since learnhd only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnhd(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnhd:

1 Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2 Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3 Set each net.inputWeights{i,j}.learnFcn to 'learnhd'. Set each net.layerWeights{i,j}.learnFcn to 'learnhd'. (Each weight learning parameter property will automatically be set to learnhd's default parameters.)

To train the network (or enable it to adapt):

1 Set net.trainParam (net.adaptParam) properties to desired values.
2 Call train (adapt).

Algorithm

learnhd calculates the weight change dW for a given neuron from the neuron's input P, output A, decay rate DR, and learning rate LR according to the Hebb with decay learning rule:

```
dw =  lr*a*p' - dr*w
```

See Also

learnh, adapt, train

learnis

Purpose

Instar weight learning function

Syntax

```
[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnis(code)
```

Description

`learnis` is the instar weight learning function.

`learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S x 1` bias vector).
- `P` - `R` x `Q` input vectors (or ones`(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be `= []`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnis`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01` - Learning rate.

`learnis(code)` return useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnis only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnis(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer i of a custom network so that it can learn with learnis:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnis'. Set each net.layerWeights{i,j}.learnFcn to 'learnis'. (Each weight learning parameter property will automatically be set to learnis's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

Algorithm

learnis calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the instar learning rule:

```
dw =  lr*a*(p'-w)
```

See Also

learnk, learnos, adapt, train

References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

learnk

Purpose

Kohonen weight learning function

Syntax

```
[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnk(code)
```

Description

`learnk` is the Kohonen weight learning function.

`learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S x 1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnk`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01` - Learning rate.

`learnk(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnk only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnk(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights of layer i of a custom network to learn with learnk:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnk'. Set each net.layerWeights{i,j}.learnFcn to 'learnk'. (Each weight learning parameter property will automatically be set to learnk's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

Algorithm

learnk calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the Kohonen learning rule:

dw = lr*(p'-w), if a ~= 0; = 0, otherwise.

See Also

learnis, learnos, adapt, train

References

Kohonen, T., *Self-Organizing and Associative Memory*, New York: Springer-Verlag, 1984.

learnlv1

Purpose LVQ1 weight learning function

Syntax

```
[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv1(code)
```

Description

`learnlv1` is the LVQ1 weight learning function.

`learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `R` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnlv1`'s learning parameter shown here with its default value.

- `LP.lr - 0.01 -` Learning rate.

`learnlv1(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- `needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P, output A, weight matrix W, and output gradient gA for a layer with a two-element input and three neurons.

We also define the learning rate LR.

```
p = rand(2,1);
w = rand(3,2);
a = compet(negdist(w,p));
gA = [-1;1; 1];
lp.lr = 0.5;
```

Since learnlv1 only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnlv1(w,p,[],[],a,[],[],[],gA,[],lp,[])
```

Network Use

You can create a standard network that uses learnlv1 with newlvq. To prepare the weights of layer i of a custom network to learn with learnlv1:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnlv1'. Set each net.layerWeights{i,j}.learnFcn to 'learnlv1'. (Each weight learning parameter property will automatically be set to learnlv1's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

Algorithm

learnlv1 calculates the weight change dW for a given neuron from the neuron's input P, output A, output gradient gA and learning rate LR, according to the LVQ1 rule, given i the index of the neuron whose output a(i) is 1:

dw(i,:) = +lr*(p-w(i,:)) if gA(i) = 0;= -lr*(p-w(i,:)) if gA(i) = -1

See Also

learnlv2, adapt, train

learnlv2

Purpose LVQ2.1 weight learning function

Syntax

```
[dW,LS] = learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv2(code)
```

Description

`learnlv2` is the `LVQ2` weight learning function.

`learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnlv1`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01` - Learning rate.
- `LP.window - 0.25` - Window size (0 to 1, typically 0.2 to 0.3).

`learnlv2(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a sample input P, output A, weight matrix W, and output gradient gA for a layer with a two-element input and three neurons.

We also define the learning rate LR.

```
p = rand(2,1);
w = rand(3,2);
n = negdist(w,p);
a = compet(n);
gA = [-1;1; 1];
lp.lr = 0.5;
```

Since learnlv2 only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnlv2(w,p,[],n,a,[],[],[],gA,[],lp,[])
```

Network Use

You can create a standard network that uses learnlv2 with newlvq.

To prepare the weights of layer i of a custom network to learn with learnlv2:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnlv2'. Set each net.layerWeights{i,j}.learnFcn to 'learnlv2'. (Each weight learning parameter property will automatically be set to learnlv2's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

Algorithm

learnlv2 implements Learning Vector Quantization 2.1, which works as follows:

For each presentation, if the winning neuron i should not have won, and the runner up j should have, and the distance di between the winning neuron and the input p is roughly equal to the distance dj from the runner up neuron to the input p according to the given window,

```
min(di/dj, dj/di) > (1-window)/(1+window)
```

then move the winning neuron i weights away from the input vector, and move the runner up neuron j weights toward the input according to:

```
dw(i,:) = - lp.lr*(p'-w(i,:))
dw(j,:) = + lp.lr*(p'-w(j,:))
```

See Also learnlv1, adapt, train

Purpose Outstar weight learning function

Syntax

```
[dW,LS] = learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnos(code)
```

Description

`learnos` is the outstar weight learning function.

`learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnos`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01 -` Learning rate.

`learnos(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnos only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnos(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnos:

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnos'. Set each net.layerWeights{i,j}.learnFcn to 'learnos'. (Each weight learning parameter property will automatically be set to learnos's default parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

Algorithm

learnos calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the outstar learning rule:

```
dw =  lr*(a-w)*p'
```

See Also

learnis, learnk, adapt, train

References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

Purpose Perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnp(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnp(code)
```

Description

`learnp` is the perceptron weight/bias learning function.

`learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or b, and `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

`learnp(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input `P` and error `E` to a layer with a two-element input and three neurons.

```
p = rand(2,1);
```

```
e = rand(3,1);
```

Since learnp only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnp([],p,[],[],[],[],e,[],[],[],[],[])
```

Network Use

You can create a standard network that uses learnp with newp.

To prepare the weights and the bias of layer i of a custom network to learn with learnp:

1. Set net.trainFcn to 'trainb'. (net.trainParam will automatically become trainb's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnp'. Set each net.layerWeights{i,j}.learnFcn to 'learnp'. Set net.biases{i}.learnFcn to 'learnp'. (Each weight and bias learning parameter property will automatically become the empty matrix since learnp has no learning parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

See newp for adaption and training examples.

Algorithm

learnp calculates the weight change dW for a given neuron from the neuron's input P and error E according to the perceptron learning rule:

```
dw =  0,  if e =  0
=  p', if e =  1
= -p', if e = -1
```

This can be summarized as:

```
dw = e*p'
```

See Also

learnpn, newp, adapt, train

References

Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C.: Spartan Press, 1961.

learnpn

Purpose Normalized perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnpn(code)
```

Description

`learnpn` is a weight and bias learning function. It can result in faster learning than `learnp` when input vectors have widely varying magnitudes.

`learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or ones`(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

`learnpn(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input `P` and error `E` to a layer with a two-element input and three neurons.

```
p = rand(2,1);
```

```
e = rand(3,1);
```

Since learnpn only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnpn([],p,[],[],[],[],e,[],[],[],[],[])
```

Network Use

You can create a standard network that uses learnpn with newp.

To prepare the weights and the bias of layer i of a custom network to learn with learnpn:

1 Set net.trainFcn to 'trainb'. (net.trainParam will automatically become trainb's default parameters.)
2 Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3 Set each net.inputWeights{i,j}.learnFcn to 'learnpn'. Set each net.layerWeights{i,j}.learnFcn to 'learnpn'. Set net.biases{i}.learnFcn to 'learnpn'. (Each weight and bias learning parameter property will automatically become the empty matrix since learnpn has no learning parameters.)

To train the network (or enable it to adapt):

1 Set net.trainParam (net.adaptParam) properties to desired values.
2 Call train (adapt).

See newp for adaption and training examples.

Algorithm

learnpn calculates the weight change dW for a given neuron from the neuron's input P and error E according to the normalized perceptron learning rule:

```
pn = p / sqrt(1 + p(1)^2 + p(2)^2) + ... + p(R)^2)
dw =  0,   if e =  0
=  pn', if e =  1
= -pn', if e = -1
```

The expression for dW can be summarized as:

```
dw = e*pn'
```

Limitations

Perceptrons do have one real limitation. The set of input vectors must be linearly separable if a solution is to be found. That is, if the input vectors with targets of 1 cannot be separated by a line or hyperplane from the input vectors associated with values of 0, the perceptron will never be able to classify them correctly.

See Also

`learnp`, `newp`, `adapt`, `train`

Purpose Self-organizing map weight learning function

Syntax

```
[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnsom(code)
```

Description

`learnsom` is the self-organizing map weight learning function.

`learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnsom`'s learning parameter, shown here with its default value.

`LP.order_lr`	`0.9`	Ordering phase learning rate.
`LP.order_steps`	`1000`	Ordering phase steps.
`LP.tune_lr`	`0.02`	Tuning phase learning rate.
`LP.tune_nd`	`1`	Tuning phase neighborhood distance.

`learnpn(code)` returns useful information for each `code` string:

'`pnames`' - Names of learning parameters.

'`pdefaults`' - Default learning parameters.

'`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input `P`, output `A`, and weight matrix `W`, for a layer with a two-element input and six neurons. We also calculate positions and distances for the neurons, which are arranged in a 2-by-3 hexagonal pattern. Then we define the four learning parameters.

```
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp.order_lr = 0.9;
lp.order_steps = 1000;
lp.tune_lr = 0.02;
lp.tune_nd = 1;
```

Since `learnsom` only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
ls = [];
[dW,ls] = learnsom(w,p,[],[],a,[],[],[],[],d,lp,ls)
```

Network Use

You can create a standard network that uses `learnsom` with `newsom`.

1. Set `net.trainFcn` to '`trainr`'. (`net.trainParam` will automatically become `trainr`'s default parameters.)
2. Set `net.adaptFcn` to '`trains`'. (`net.adaptParam` will automatically become `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to '`learnsom`'. Set each `net.layerWeights{i,j}.learnFcn` to '`learnsom`'. Set `net.biases{i}.learnFcn` to '`learnsom`'. (Each weight learning parameter property will automatically be set to `learnsom`'s default parameters.)

To train the network (or enable it to adapt):

1. Set `net.trainParam` (`net.adaptParam`) properties to desired values.

2 Call `train` (`adapt`).

Algorithm

`learnsom` calculates the weight change `dW` for a given neuron from the neuron's input `P`, activation `A2`, and learning rate `LR`:

```
dw =  lr*a2*(p'-w)
```

where the activation `A2` is found from the layer output `A` and neuron distances `D` and the current neighborhood size `ND`:

```
a2(i,q) = 1,   if a(i,q) = 1
        = 0.5, if a(j,q) = 1 and D(i,j) <= nd
        = 0,   otherwise
```

The learning rate `LR` and neighborhood size `NS` are altered through two phases: an ordering phase and a tuning phase.

The ordering phases lasts as many steps as `LP.order_steps`. During this phase `LR` is adjusted from `LP.order_lr` down to `LP.tune_lr`, and `ND` is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase `LR` decreases slowly from `LP.tune_lr` and `ND` is always set to `LP.tune_nd`. During this phase the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

See Also

`adapt`, `train`

learnwh

Purpose

Widrow-Hoff weight/bias learning function

Syntax

```
[dW,LS] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnwh(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnwh(code)
```

Description

`learnwh` is the Widrow-Hoff weight/bias learning function, and is also known as the delta or least mean squared (LMS) rule.

`learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` - `S` x `R` weight matrix (or b, and `S` x `1` bias vector).
- `P` - `R` x `Q` input vectors (or `ones(1,Q)`).
- `Z` - `S` x `Q` weighted input vectors.
- `N` - `S` x `Q` net input vectors.
- `A` - `S` x `Q` output vectors.
- `T` - `S` x `Q` layer target vectors.
- `E` - `S` x `Q` layer error vectors.
- `gW` - `S` x `R` weight gradient with respect to performance.
- `gA` - `S` x `Q` output gradient with respect to performance.
- `D` - `S` x `S` neuron distances.
- `LP` - Learning parameters, none, `LP = []`.
- `LS` - Learning state, initially should be = `[]`.

and returns,

- `dW` - `S` x `R` weight (or bias) change matrix.
- `LS` - New learning state.

Learning occurs according to `learnwh`'s learning parameter shown here with its default value.

- `LP.lr - 0.01` - Learning rate.

`learnwh(code)` returns useful information for each `code` string:

- '`pnames`' - Names of learning parameters.
- '`pdefaults`' - Default learning parameters.
- '`needg`' - Returns 1 if this function uses `gW` or `gA`.

Examples

Here we define a random input P and error E to a layer with a two-element input and three neurons. We also define the learning rate LR learning parameter.

```
p = rand(2,1);
e = rand(3,1);
lp.lr = 0.5;
```

Since learnwh only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnwh([],p,[],[],[],[],e,[],[],[],lp,[])
```

Network Use

You can create a standard network that uses learnwh with newlin.

To prepare the weights and the bias of layer i of a custom network to learn with learnwh:

1. Set net.trainFcn to 'trainb'. net.trainParam will automatically become trainb's default parameters.
2. Set net.adaptFcn to 'trains'. net.adaptParam will automatically become trains's default parameters.
3. Set each net.inputWeights{i,j}.learnFcn to 'learnwh'. Set each net.layerWeights{i,j}.learnFcn to 'learnwh'. Set net.biases{i}.learnFcn to 'learnwh'.

Each weight and bias learning parameter property will automatically be set to learnwh's default parameters.

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train(adapt).

See newlin for adaption and training examples.

Algorithm

learnwh calculates the weight change dW for a given neuron from the neuron's input P and error E, and the weight (or bias) learning rate LR, according to the Widrow-Hoff learning rule:

```
dw = lr*e*pn'
```

See Also `newlin, adapt, train`

References

Widrow, B., and M. E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, pp. 96-104, 1960.

Widrow B. and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

Purpose Link distance function

Syntax

```
d = linkdist(pos)
```

Description

`linkdist` is a layer distance function used to find the distances between the layer's neurons given their positions.

`linkdist`(pos) takes one argument,

  `pos - N` x `S` matrix of neuron positions.

and returns the `S` x `S` matrix of distances.

Examples

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = linkdist(pos)
```

Network Use

You can create a standard network that uses `linkdist` as a distance function by calling `newsom`.

To change a network so that a layer's topology uses `linkdist`, set `net.layers{i}.distanceFcn` to '`linkdist`'.

In either case, call `sim` to simulate the network with `dist`. See `newsom` for training and adaption examples.

Algorithm

The link distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is:

```
Dij = 0, if i==j
= 1, if (sum((Pi-Pj).^2)).^0.5 is <= 1
= 2, if k exists, Dik = Dkj = 1
= 3, if k1, k2 exist, Dik1 = Dk1k2 = Dk2j = 1
= N, if k1..kN exist, Dik1 = Dk1k2 = ...= DkNj = 1
= S, if none of the above conditions apply.
```

See Also

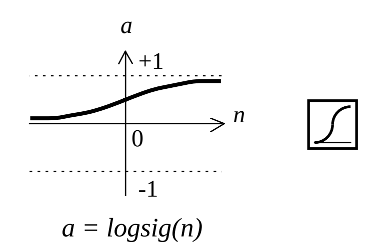
`sim`, `dist`, `mandist`

logsig

Purpose

Log sigmoid transfer function

Graph and Symbol



Log-Sigmoid Transfer Function

Syntax

```
A = logsig(N)
info = logsig(code)
```

Description

`logsig` is a transfer function. Transfer functions calculate a layer's output from its net input.

`logsig(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns each element of `N` squashed between 0 and 1.

`logsig(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here is the code to create a plot of the `logsig` transfer function.

```
n = -5:0.1:5;
a = logsig(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `logsig` by calling `newff` or `newcf`.

To change a network so a layer uses `logsig`, set `net.layers{i}.transferFcn` to '`logsig`'.

In either case, call `sim` to simulate the network with `purelin`.

See `newff` or `newcf` for simulation examples.

Algorithm

```
logsig(n) = 1 / (1 + exp(-n))
```

See Also

`sim`, `dlogsig`, `tansig`

mae

Purpose Mean absolute error performance function

Syntax

```
perf = mae(E,X,PP)
perf = mae(E,net,PP)
info = mae(code)
```

Description

`mae` is a network performance function.

`mae(E,X,PP)` takes from one to three arguments,

- `E` - Matrix or cell array of error vector(s).
- `X` - Vector of all weight and bias values (ignored).
- `PP` - Performance parameters (ignored).

and returns the mean absolute error.

The errors `E` can be given in cell array form,

- `E` - `Nt` x `TS` cell array, each element `E{i,ts}` is a `Vi x Q` matrix or`[]`.

or as a matrix,

- `E` - `(sum of Vi)` x `Q` matrix

where

- `Nt` = `net.numTargets`
- `TS` = Number of time steps
- `Q` = Batch size
- `Vi` = `net.targets{i}.size`

`mae(E,net,PP)` can take an alternate argument to `X`,

- `net` - Neural network from which `X` can be obtained (ignored).

`mae(code)` returns useful information for each `code` string:

- '`deriv`' - Name of derivative function.
- '`name`' - Full name.
- '`pnames`' - Names of training parameters.
- '`pdefaults`' - Default training parameters.

Examples

Here a perceptron is created with a 1-element input ranging from -10 to 10, and one neuron.

```
net = newp([-10 10],1);
```

Here the network is given a batch of inputs P. The error is calculated by subtracting the output A from target T. Then the mean absolute error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
perf = mae(e)
```

Note that mae can be called with only one argument because the other arguments are ignored. mae supports those arguments to conform to the standard performance function argument list.

Network Use

You can create a standard network that uses mae with newp.

To prepare a custom network to be trained with mae, set net.performFcn to 'mae'. This will automatically set net.performParam to the empty matrix [], as mae has no performance parameters.

In either case, calling train or adapt will result in mae being used to calculate performance.

See newp for examples.

See Also

mse, msereg, dmae

mandist

Purpose Manhattan distance weight function

Syntax

```
Z = mandist(W,P)
df = mandist('deriv')
D = mandist(pos);
```

Description `mandist` is the Manhattan distance weight function. Weight functions apply weights to an input to get weighted inputs.

`mandist(W,P)` takes these inputs,

`W` - `S` x `R` weight matrix.

`P` - `R` x `Q` matrix of `Q` input (column) vectors.

and returns the `S` x `Q` matrix of vector distances.

`mandist('deriv')` returns `''` because `mandist` does not have a derivative function.

`mandist` is also a layer distance function, which can be used to find the distances between neurons in a layer.

`mandist(pos)` takes one argument,

`pos` - An `S` row matrix of neuron positions.

and returns the `S` x `S` matrix of distances.

Examples Here we define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = mandist(W,P)
```

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
D = mandist(pos)
```

Network Use You can create a standard network that uses `mandist` as a distance function by calling `newsom`.

To change a network so an input weight uses `mandist`, set `net.inputWeight{i,j}.weightFcn` to '`mandist` . For a layer weight, set `net.inputWeight{i,j}.weightFcn` to '`mandist`'.

To change a network so a layer's topology uses `mandist`, set `net.layers{i}.distanceFcn` to '`mandist`'.

In either case, call `sim` to simulate the network with `dist`. See `newpnn` or `newgrnn` for simulation examples.

Algorithm

The Manhattan distance `D` between two vectors `X` and `Y` is:

```
D = sum(abs(x-y))
```

See Also

`sim`, `dist`, `linkdist`

maxlinlr

Purpose

Maximum learning rate for a linear layer

Syntax

```
lr = maxlinlr(P)
lr = maxlinlr(P,'bias')
```

Description

`maxlinlr` is used to calculate learning rates for `newlin`.

`maxlinlr(P)` takes one argument,

`P - R x Q` matrix of input vectors.

and returns the maximum learning rate for a linear layer without a bias that is to be trained only on the vectors in `P`.

`maxlinlr(P,'bias')` returns the maximum learning rate for a linear layer with a bias.

Examples

Here we define a batch of four two-element input vectors and find the maximum learning rate for a linear layer with a bias.

```
P = [1 2 -4 7; 0.1 3 10 6];
lr = maxlinlr(P,'bias')
```

See Also

`linnet, newlin, newlind`

Purpose

Midpoint weight initialization function

Syntax

```
W = midpoint(S,PR)
```

Description

`midpoint` is a weight initialization function that sets weight (row) vectors to the center of the input ranges.

`midpoint(S,PR)` takes two arguments,

- `S` - Number of rows (neurons).
- `PR` - `R` x 2 matrix of input value ranges = `[Pmin Pmax]`.

and returns an `S` x `R` matrix with rows set to `(Pmin+Pmax)'/2`.

Examples

Here initial weight values are calculated for a 5 neuron layer with input elements ranging over `[0 1]` and `[-2 2]`.

```
W = midpoint(5,[0 1; -2 2])
```

Network Use

You can create a standard network that uses `midpoint` to initialize weights by calling `newc`.

To prepare the weights and the bias of layer `i` of a custom network to initialize with `midpoint`:

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set each `net.inputWeights{i,j}.initFcn` to '`midpoint`'. Set each `net.layerWeights{i,j}.initFcn` to '`midpoint`';

To initialize the network call `init`.

See Also

`initwb`, `initlay`, `init`

minmax

Purpose Ranges of matrix rows

Syntax

```
pr = minmax(p)
```

Description

`minmax(P)` takes one argument,

`P` - `R` x `Q` matrix.

and returns the `R` x 2 matrix `PR` of minimum and maximum values for each row of `P`.

Examples

```
P = [0 1 2; -1 -2 -0.5]
pr = minmax(P)
```

Purpose Mean squared error performance function

Syntax

```
perf = mse(E,X,PP)
perf = mse(E,net,PP)
info = mse(code)
```

Description `mse` is a network performance function. It measures the network's performance according to the mean of squared errors.

`mse(E,X,PP)` takes from one to three arguments,

- `E` - Matrix or cell array of error vector(s).
- `X` - Vector of all weight and bias values (ignored).
- `PP` - Performance parameters (ignored).

and returns the mean squared error.

`mse(E,net,PP)` can take an alternate argument to `X`,

- `net` - Neural network from which `X` can be obtained (ignored).

`mse(code)` returns useful information for each `code` string:

- '`deriv`' - Name of derivative function.
- '`name`' - Full name.
- '`pnames`' - Names of training parameters.
- '`pdefaults`' - Default training parameters.

Examples Here a two-layer feed-forward network is created with a 1-element input ranging from -10 to 10, four hidden `tansig` neurons, and one `purelin` output neuron.

```
net = newff([-10 10],[4 1],{'tansig','purelin'});
```

Here the network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the mean squared error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
```

```
perf = mse(e)
```

Note that mse can be called with only one argument because the other arguments are ignored. mse supports those ignored arguments to conform to the standard performance function argument list.

Network Use

You can create a standard network that uses mse with newff, newcf, or newelm.

To prepare a custom network to be trained with mse, set net.performFcn to 'mse'. This will automatically set net.performParam to the empty matrix [], as mse has no performance parameters.

In either case, calling train or adapt will result in mse being used to calculate performance.

See newff or newcf for examples.

See Also

msereg, mae, dmse

Purpose Mean squared error with regularization performance function

Syntax

```
perf = msereg(E,X,PP)
perf = msereg(E,net,PP)
info = msereg(code)
```

Description

`msereg` is a network performance function. It measures network performance as the weight sum of two factors: the mean squared error and the mean squared weight and bias values.

`msereg(E,X,PP)` takes from three arguments,

- `E` - Matrix or cell array of error vector(s).
- `X` - Vector of all weight and bias values.
- `PP` - Performance parameter.

where `PP` defines one performance parameters,

- `PP.ratio` - Relative importance of errors vs. weight and bias values.

and returns the sum of mean squared errors (times `PP.ratio`) with the mean squared weight and bias values (times `1-PP.ratio`).

The errors `E` can be given in cell array form,

- `E` - `Nt` x `TS` cell array, each element `E{i,ts}` is an `Vi` x `Q` matrix or `[]`.

or as a matrix,

- `E` - (sum of `Vi`) x `Q` matrix

where

- `Nt = net.numTargets`
- `TS` = Number of time steps
- `Q` = Batch size
- `Vi = net.targets{i}.size`

`msereg(E,net)` takes an alternate argument to `X` and `PP`,

- `net` - Neural network from which `X` and `PP` can be obtained.

msereg(code) returns useful information for each code string:

'deriv' - Name of derivative function.

'name' - Full name.

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Examples

Here a two-layer feed-forward is created with a one-element input ranging from -2 to 2, four hidden tansig neurons, and one purelin output neuron.

```
net = newff([-2 2],[4 1]
{'tansig','purelin'},'trainlm','learngdm','msereg');
```

Here the network is given a batch of inputs P. The error is calculated by subtracting the output A from target T. Then the mean squared error is calculated using a ratio of 20/(20+1). (Errors are 20 times as important as weight and bias values).

```
p = [-2 -1 0 1 2];
t = [0 1 1 1 0];
y = sim(net,p)
e = t-y
net.performParam.ratio = 20/(20+1);
perf = msereg(e,net)
```

Network Use

You can create a standard network that uses msereg with newff, newcf, or newelm.

To prepare a custom network to be trained with msereg, set net.performFcn to 'msereg'. This will automatically set net.performParam to msereg's default performance parameters.

In either case, calling train or adapt will result in msereg being used to calculate performance.

See newff or newcf for examples.

See Also

mse, mae, dmsereg

negdist

Purpose Negative distance weight function

Syntax

```
Z = negdist(W,P)
df = negdist('deriv')
```

Description `negdist` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`negdist(W,P)` takes these inputs,

- `W - S x R` weight matrix.
- `P - R x Q` matrix of `Q` input (column) vectors.

and returns the `S x Q` matrix of negative vector distances.

`negdist('deriv')` returns `''` because `negdist` does not have a derivative function.

Examples Here we define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = negdist(W,P)
```

Network Use You can create a standard network that uses `negdist` by calling `newc` or `newsom`.

To change a network so an input weight uses `negdist`, set `net.inputWeight{i,j}.weightFcn` to 'negdist . For a layer weight set `net.inputWeight{i,j}.weightFcn` to 'negdist .

In either case, call `sim` to simulate the network with `negdist`. See `newc` or `newsom` for simulation examples.

Algorithm `negdist` returns the negative Euclidean distance:

```
z = -sqrt(sum(w-p)^2)
```

See Also `sim, dotprod, dist`

netprod

Purpose Product net input function

Syntax

```
N = netprod(Z1,Z2,...,Zn)
df = netprod('deriv')
```

Description

netprod is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

netprod(Z1,Z2,...,Zn) takes,

Zi - S x Q matrices.

and returns an element-wise sum of Zi's.

netprod('deriv') returns netprod's derivative function.

Examples

Here netprod combines two sets of weighted input vectors (which we have defined ourselves).

```
z1 = [1 2 4;3 4 1];
z2 = [-1 2 2; -5 -6 1];
n = netprod(Z1,Z2)
```

Here netprod combines the same weighted inputs with a bias vector. Because Z1 and Z2 each contain three concurrent vectors, three concurrent copies of B must be created with concur so that all sizes match up.

```
b = [0; -1];
n = netprod(z1,z2,concur(b,3))
```

Network Use

You can create a standard network that uses netprod by calling newpnn or newgrnn.

To change a network so that a layer uses netprod, set net.layers{i}.netInputFcn to 'netprod'.

In either case, call sim to simulate the network with netprod. See newpnn or newgrnn for simulation examples.

See Also

sim, dnetprod, netsum, concur

Purpose Sum net input function

Syntax

```
N = netsum(Z1,Z2,...,Zn)
df = netsum('deriv')
```

Description netsum is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

netsum(Z1,Z2,...,Zn) takes any number of inputs,

Zi - S x Q matrices,

and returns N, the element-wise sum of Zi's.

netsum('deriv') returns netsum's derivative function.

Examples Here netsum combines two sets of weighted input vectors (which we have defined ourselves).

```
z1 = [1 2 4;3 4 1];
z2 = [-1 2 2; -5 -6 1];
n = netsum(Z1,Z2)
```

Here netsum combines the same weighted inputs with a bias vector. Because Z1 and Z2 each contain three concurrent vectors, three concurrent copies of B must be created with concur so that all sizes match up.

```
b = [0; -1];
n = netsum(z1,z2,concur(b,3))
```

Network Use You can create a standard network that uses netsum by calling newp or newlin.

To change a network so a layer uses netsum, set net.layers{i}.netInputFcn to 'netsum'.

In either case, call sim to simulate the network with netsum. See newp or newlin for simulation examples.

See Also sim, dnetprod, netprod, concur

network

Purpose Create a custom neural network

Syntax

```
net = network
net = network(numInputs,numLayers,biasConnect,inputConnect,
layerConnect,outputConnect,targetConnect)
```

To Get Help Type `help network/network`

Description

`network` creates new custom networks. It is used to create networks that are then customized by functions such as `newp`, `newlin`, `newff`, etc.

`network` takes these optional arguments (shown with default values):

`numInputs`	-	Number of inputs, 0.
`numLayers`	-	Number of layers, 0.
`biasConnect`	-	`numLayers`-by-1 Boolean vector, zeros.
`inputConnect`	-	`numLayers`-by-`numInputs` Boolean matrix, zeros.
`layerConnect`	-	`numLayers`-by-`numLayers` Boolean matrix, zeros.
`outputConnect`	-	1-by-`numLayers` Boolean vector, zeros.
`targetConnect`	-	1-by-`numLayers` Boolean vector, zeros.

and returns,

`net` - New network with the given property values.

Properties

Architecture properties:

`net.numInputs`: 0 or a positive integer.

Number of inputs.

`net.numLayers`: 0 or a positive integer.

Number of layers.

`net.biasConnect`: `numLayer`-by-1 Boolean vector.

If `net.biasConnect(i)` is 1, then the layer `i` has a bias and `net.biases{i}` is a structure describing that bias.

`net.inputConnect`: `numLayer`-by-`numInputs` Boolean vector.

If `net.inputConnect(i,j)` is 1, then layer `i` has a weight coming from input `j` and `net.inputWeights{i,j}` is a structure describing that weight.

`net.layerConnect`: numLayer-by-numLayers Boolean vector.

If `net.layerConnect(i,j)` is 1, then layer `i` has a weight coming from layer `j` and `net.layerWeights{i,j}` is a structure describing that weight.

`net.outputConnect`: 1-by-`numLayers` Boolean vector.

If `net.outputConnect(i)` is 1, then the network has an output from layer `i` and `net.outputs{i}` is a structure describing that output.

`net.targetConnect`: 1-by-`numLayers` Boolean vector.

If `net.outputConnect(i)` is 1, then the network has a target from layer `i` and `net.targets{i}` is a structure describing that target.

`net.numOutputs`: 0 or a positive integer. Read only.

Number of network outputs according to `net.outputConnect`.

`net.numTargets`: 0 or a positive integer. Read only.

Number of targets according to `net.targetConnect`.

`net.numInputDelays`: 0 or a positive integer. Read only.

Maximum input delay according to all `net.inputWeight{i,j}`.delays.

`net.numLayerDelays`: 0 or a positive number. Read only.

Maximum layer delay according to all `net.layerWeight{i,j}`.delays.

Subobject structure properties:

`net.inputs`: `numInputs`-by-1 cell array.

`net.inputs{i}` is a structure defining input `i`.

`net.layers`: `numLayers`-by-1 cell array.

`net.layers{i}` is a structure defining layer `i`.

`net.biases`: `numLayers`-by-1 cell array.

If `net.biasConnect(i)` is 1, then `net.biases{i}` is a structure defining the bias for layer `i`.

`net.inputWeights`: `numLayers`-by-`numInputs` cell array.

If `net.inputConnect(i,j)` is 1, then `net.inputWeights{i,j}` is a structure defining the weight to layer `i` from input `j`.

`net.layerWeights`: `numLayers`-by-`numLayers` cell array.

If `net.layerConnect(i,j)` is 1, then `net.layerWeights{i,j}` is a structure defining the weight to layer `i` from layer `j`.

`net.outputs`: 1-by-`numLayers` cell array.

If `net.outputConnect(i)` is 1, then `net.outputs{i}` is a structure defining the network output from layer `i`.

`net.targets`: 1-by-`numLayers` cell array.

If `net.targetConnect(i)` is 1, then `net.targets{i}` is a structure defining the network target to layer `i`.

Function properties:

`net.adaptFcn`: name of a network adaption function or `''`.

`net.initFcn`: name of a network initialization function or `''`.

`net.performFcn`: name of a network performance function or `''`.

`net.trainFcn`: name of a network training function or `''`.

Parameter properties:

`net.adaptParam`: network adaption parameters.

`net.initParam`: network initialization parameters.

`net.performParam`: network performance parameters.

`net.trainParam`: network training parameters.

Weight and bias value properties:

`net.IW`: `numLayers`-by-`numInputs` cell array of input weight values.

`net.LW`: `numLayers`-by-`numLayers` cell array of layer weight values.

`net.b`: `numLayers`-by-1 cell array of bias values.

Other properties:

`net.userdata`: structure you can use to store useful values.

Examples

Here is the code to create a network without any inputs and layers, and then set its number of inputs and layer to 1 and 2 respectively.

```
net = network
net.numInputs = 1
net.numLayers = 2
```

Here is the code to create the same network with one line of code.

```
net = network(1,2)
```

Here is the code to create a 1 input, 2 layer, feed-forward network. Only the first layer will have a bias. An input weight will connect to layer 1 from input 1. A layer weight will connect to layer 2 from layer 1. Layer 2 will be a network output, and have a target.

```
net = network(1,2,[1;0],[1; 0],[0 0; 1 0],[0 1],[0 1])
```

We can then see the properties of subobjects as follows:

```
net.inputs{1}
net.layers{1}, net.layers{2}
net.biases{1}
net.inputWeights{1,1}, net.layerWeights{2,1}
net.outputs{2}
net.targets{2}
```

We can get the weight matrices and bias vector as follows:

```
net.iw.{1,1}, net.iw{2,1}, net.b{1}
```

We can alter the properties of any of these subobjects. Here we change the transfer functions of both layers:

```
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'logsig';
```

Here we change the number of elements in input 1 to 2, by setting each element's range:

```
net.inputs{1}.range = [0 1; -1 1];
```

Next we can simulate the network for a two-element input vector:

```
p = [0.5; -0.1];
y = sim(net,p)
```

See Also sim

Purpose

Create a competitive layer

Syntax

```
net = newc
net = newc(PR,S,KLR,CLR)
```

Description

Competitive layers are used to solve classification problems.

`net = newc` creates a new network with a dialog box.

`net = newc(PR,S,KLR,CLR)` takes these inputs,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `S` - Number of neurons.
- `KLR` - Kohonen learning rate, default = 0.01.
- `CLR` - Conscience learning rate, default = 0.001.

and returns a new competitive layer.

Properties

Competitive layers consist of a single layer with the `negdist` weight function, `netsum` net input function, and the `compet` transfer function.

The layer has a weight from the input, and a bias.

Weights and biases are initialized with `midpoint` and `initcon`.

Adaption and training are done with `trains` and `trainr`, which both update weight and bias values with the `learnk` and `learncon` learning functions.

Examples

Here is a set of four two-element vectors `P`.

```
P = [.1 .8  .1 .9; .2 .9 .1 .8];
```

A competitive layer can be used to divide these inputs into two classes. First a two neuron layer is created with two input elements ranging from 0 to 1, then it is trained.

```
net = newc([0 1; 0 1],2);
net = train(net,P);
```

The resulting network can then be simulated and its output vectors converted to class indices.

```
Y = sim(net,P)
```

```
Yc = vec2ind(Y)
```

See Also

sim, init, adapt, train, trains, trainr, newcf

Purpose

Create a trainable cascade-forward backpropagation network

Syntax

```
net = newcf
net = newcf(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

Description

`net = newcf` creates a new network with a dialog box.

`newcf(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `Si` - Size of ith layer, for `Nl` layers.
- `TFi` - Transfer function of ith layer, default = '`tansig`'.
- `BTF` - Backpropagation network training function, default = '`traingd`'.
- `BLF` - Backpropagation weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns an `N` layer cascade-forward backprop network.

The transfer functions `TFi` can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

Caution: `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an “out-of-memory” error when training try doing one of these:

---

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Examples

Here is a problem consisting of inputs `P` and targets `T` that we would like to solve with a network.

```
P = [0 1 2 3 4 5 6 7 8 9 10];
T = [0 1 2 3 4 3 2 1 2 3 4];
```

Here a two-layer cascade-forward network is created. The network's input ranges from [0 to 10]. The first layer has five `tansig` neurons, the second layer has one `purelin` neuron. The `trainlm` network training function is to be used.

```
net = newcf([0 10],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated and its output plotted against the targets.

```
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Here the network is trained for 50 epochs. Again the network's output is plotted.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Algorithm

Cascade-forward networks consist of `Nl` layers using the `dotprod` weight function, `netsum` net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has weights coming from the input and all previous layers. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with `initnw`.

Adaption is done with `trains`, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

See Also

`newff`, `newelm`, `sim`, `init`, `adapt`, `train`, `trains`

Purpose Create an Elman backpropagation network

Syntax

```
net = newelm
net = newelm(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

Description

`net = newelm` creates a new network with a dialog box.

`newelm(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes several arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `Si` - Size of ith layer, for `Nl` layers.
- `TFi` - Transfer function of ith layer, default = '`tansig`'.
- `BTF` - Backpropagation network training function, default = '`traingdx`'.
- `BLF` - Backpropagation weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns an Elman network.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

Caution: `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an “out-of-memory” error when training try doing one of these:

---

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Examples

Here is a series of Boolean inputs P, and another sequence T, which is 1 wherever P has had two 1's in a row.

```
P = round(rand(1,20));
T = [0 (P(1:end-1)+P(2:end) == 2)];
```

We would like the network to recognize whenever two 1's occur in a row. First we arrange these values as sequences.

```
Pseq = con2seq(P);
Tseq = con2seq(T);
```

Next we create an Elman network whose input varies from 0 to 1, and has five hidden neurons and 1 output.

```
net = newelm([0 1],[10 1],{'tansig','logsig'});
```

Then we train the network with a mean squared error goal of 0.1, and simulate it.

```
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq)
```

Algorithm

Elman networks consist of `Nl` layers using the `dotprod` weight function, `netsum` net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers except the last have a recurrent weight. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with `initnw`.

Adaption is done with `trains`, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

See Also

`newff, newcf, sim, init, adapt, train, trains`

newff

Purpose Create a feed-forward backpropagation network

Syntax

```
net = newff
net = newff(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

Description

`net = newff` creates a new network with a dialog box.

`newff(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `Si` - Size of ith layer, for `Nl` layers.
- `TFi` - Transfer function of ith layer, default = '`tansig`'.
- `BTF` - Backpropagation network training function, default = '`traingdx`'.
- `BLF` - Backpropagation weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns an `N` layer feed-forward backprop network.

The transfer functions `TFi` can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

Caution: `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

---

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Examples

Here is a problem consisting of inputs `P` and targets `T` that we would like to solve with a network.

```
P = [0 1 2 3 4 5 6 7 8 9 10];
T = [0 1 2 3 4 3 2 1 2 3 4];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has five `tansig` neurons, the second layer has one `purelin` neuron. The `trainlm` network training function is to be used.

```
net = newff([0 10],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated and its output plotted against the targets.

```
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Here the network is trained for 50 epochs. Again the network's output is plotted.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Algorithm

Feed-forward networks consist of `Nl` layers using the `dotprod` weight function, `netsum` net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with `initnw`.

Adaption is done with `trains`, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

See Also

`newcf`, `newelm`, `sim`, `init`, `adapt`, `train`, `trains`

Purpose Create a feed-forward input-delay backpropagation network

Syntax

```
net = newfftd
net = newfftd(PR,ID,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

Description

`net = newfftd` creates a new network with a dialog box.

`newfftd(PR,ID,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `ID` - Input delay vector.
- `Si` - Size of ith layer, for `Nl` layers.
- `TFi` - Transfer function of ith layer, default = '`tansig`'.
- `BTF` - Backprop network training function, default = '`traingdx`'.
- `BLF` - Backprop weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns an `N` layer feed-forward backprop network.

The transfer functions `TFi` can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

Caution: `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

---

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as mse or msereg.

Examples

Here is a problem consisting of an input sequence P and target sequence T that can be solved by a network with one delay.

```
P = {1  0 0 1 1  0 1  0 0 0 0 1 1  0 0 1};
T = {1 -1 0 1 0 -1 1 -1 0 0 0 1 0 -1 0 1};
```

Here a two-layer feed-forward network is created with input delays of 0 and 1. The network's input ranges from [0 to 1]. The first layer has five tansig neurons, the second layer has one purelin neuron. The trainlm network training function is to be used.

```
net = newfftd([0 1],[0 1],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated.

```
Y = sim(net,P)
```

Here the network is trained for 50 epochs. Again the network's output is calculated.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P)
```

Algorithm

Feed-forward networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer functions.

The first layer has weights coming from the input with the specified input delays. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

See Also

newcf, newelm, sim, init, adapt, train, trains

Purpose

Design a generalized regression neural network (grnn)

Syntax

```
net = newgrnn
net = newgrnn(P,T,spread)
```

Description

`net = newgrnn` creates a new network with a dialog box.

Generalized regression neural networks are a kind of radial basis network that is often used for function approximation. `grnn` s can be designed very quickly.

`newgrnn(P,T,spread)` takes three inputs,

- `P` - `R` x `Q` matrix of `Q` input vectors.
- `T` - `S` x `Q` matrix of `Q` target class vectors.
- `spread` - Spread of radial basis functions, default = 1.0.

and returns a new generalized regression neural network.

The larger the `spread`, is the smoother the function approximation will be. To fit data very closely, use a spread smaller than the typical distance between input vectors. To fit the data more smoothly, use a larger `spread`.

Properties

`newgrnn` creates a two-layer network. The first layer has `radbas` neurons, calculates weighted inputs with `dist` and net input with `netprod`. The second layer has `purelin` neurons, calculates weighted input with `normprod` and net inputs with `netsum`. Only the first layer has biases.

`newgrnn` sets the first layer weights to `P'`, and the first layer biases are all set to 0.8326/`spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- `spread`. The second layer weights `W2` are set to `T`.

Examples

Here we design a radial basis network given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newgrnn(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

See Also

`sim`, `newrb`, `newrbe`, `newpnn`

References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, pp. 155-61, 1993.

newhop

Purpose

Create a Hopfield recurrent network

Syntax

```
net = newhop
net = newhop(T)
```

Description

Hopfield networks are used for pattern recall.

`net = newhop` creates a new network with a dialog box.

`newhop(T)` takes one input argument,

`T` - `R` x `Q` matrix of `Q` target vectors. (Values must be +1 or -1.)

and returns a new Hopfield recurrent neural network with stable points at the vectors in `T`.

Properties

Hopfield networks consist of a single layer with the `dotprod` weight function, `netsum` net input function, and the `satlins` transfer function.

The layer has a recurrent weight from itself and a bias.

Examples

Here we create a Hopfield network with two three-element stable points `T`.

```
T = [-1 -1 1; 1 -1 1]';
net = newhop(T);
```

Below we check that the network is stable at these points by using them as initial layer delay conditions. If the network is stable we would expect that the outputs `Y` will be the same. (Since Hopfield networks have no inputs, the second argument to `sim` is `Q = 2` when using matrix notation).

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

To see if the network can correct a corrupted vector, run the following code, which simulates the Hopfield network for five time steps. (Since Hopfield networks have no inputs, the second argument to `sim` is `{Q TS} = [1 5]` when using cell array notation.)

```
Ai = {[-0.9; -0.8; 0.7]};
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

If you run the above code, `Y{1}` will equal `T(:,1)` if the network has managed to convert the corrupted vector `Ai` to the nearest target vector.

Algorithm

Hopfield networks are designed to have stable layer outputs as defined by user-supplied targets. The algorithm minimizes the number of unwanted stable points.

See Also

`sim`, `satlins`

References

Li, J., A. N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405-1422, November 1989.

Purpose Create a linear layer

Syntax

```
net = newlin
net = newlin(PR,S,ID,LR)
```

Description Linear layers are often used as adaptive filters for signal processing and prediction.

`net = newlin` creates a new network with a dialog box.

`newlin(PR,S,ID,LR)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `S` - Number of elements in the output vector.
- `ID` - Input delay vector, default = [0].
- `LR` - Learning rate, default = 0.01.

and returns a new linear layer.

`net = newlin(PR,S,0,P)` takes an alternate argument,

- `P` - Matrix of input vectors.

and returns a linear layer with the maximum stable learning rate for learning with inputs `P`.

Examples This code creates a single input (range of [-1 1] linear layer with one neuron, input delays of 0 and 1, and a learning rate of 0.01. It is simulated for an input sequence `P1`.

```
net = newlin([-1 1],1,[0 1],0.01);
P1 = {0 -1 1 1 0 -1 1 0 0 1};
Y = sim(net,P1)
```

Here targets `T1` are defined and the layer adapts to them. (Since this is the first call to `adapt`, the default input delay conditions are used.)

```
T1 = {0 -1 0 2 1 -1 0 1 0 1};
[net,Y,E,Pf] = adapt(net,P1,T1); Y
```

Here the linear layer continues to adapt for a new sequence using the previous final conditions `PF` as initial conditions.

```
P2 = {1 0 -1 -1 1 1 1 0 -1};
T2 = {2 1 -1 -2 0 2 2 1 0};
[net,Y,E,Pf] = adapt(net,P2,T2,Pf); Y
```

Here we initialize the layer's weights and biases to new values.

```
net = init(net);
```

Here we train the newly initialized layer on the entire sequence for 200 epochs to an error goal of 0.1.

```
P3 = [P1 P2];
T3 = [T1 T2];
net.trainParam.epochs = 200;
net.trainParam.goal = 0.1;
net = train(net,P3,T3);
Y = sim(net,[P1 P2])
```

Algorithm

Linear layers consist of a single layer with the `dotprod` weight function, `netsum` net input function, and `purelin` transfer function.

The layer has a weight from the input and a bias.

Weights and biases are initialized with `initzero`.

Adaption and training are done with `trains` and `trainb`, which both update weight and bias values with `learnwh`. Performance is measured with `mse`.

See Also

`newlind`, `sim`, `init`, `adapt`, `train`, `trains`, `trainb`

Purpose

Design a linear layer

Syntax

```
net = newlind
net = newlind(P,T,Pi)
```

Description

`net = newlind` creates a new network with a dialog box.

`newlind(P,T,Pi)` takes these input arguments,

- `P` - `R` x `Q` matrix of `Q` input vectors.
- `T` - `S` x `Q` matrix of `Q` target class vectors.
- `Pi`- `1` x `ID` cell array of initial input delay states,

  where each element `Pi{i,k}` is an `RixQ` matrix, default = []

and returns a linear layer designed to output `T` (with minimum sum square error) given input `P`.

`newlind(P,T,Pi)` can also solve for linear networks with input delays and multiple inputs and layers by supplying input and target data in cell array form:

- `P` - `NixTS` cell array, each element `P{i,ts}` is an `Ri` x `Q` input matrix.
- `T` - `NtxTS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.
- `Pi` - `NixID` cell array, each element `Pi{i,k}` is an `Ri` x `Q` matrix, default = [].

returns a linear network with `ID` input delays, `Ni` network inputs, `Nl` layers, and designed to output `T` (with minimum sum square error) given input `P`.

Examples

We would like a linear layer that outputs `T` given `P` for the following definitions.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
```

Here we use `newlind` to design such a network and check its response.

```
net = newlind(P,T);
Y = sim(net,P)
```

We would like another linear layer that outputs the sequence `T` given the sequence `P` and two initial input delay states `Pi`.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5.0 6.1 4.0 6.0 6.9 8.0};
net = newlind(P,T,Pi);
Y = sim(net,P,Pi)
```

We would like a linear network with two outputs `Y1` and `Y2` that generate sequences `T1` and `T2`, given the sequences `P1` and `P2` with three initial input delay states `Pi1` for input 1, and three initial delays states `Pi2` for input 2.

```
P1 = {1 2 1 3 3 2}; Pi1 = {1 3 0};
P2 = {1 2 1 1 2 1}; Pi2 = {2 1 2};
T1 = {5.0 6.1 4.0 6.0 6.9 8.0};
T2 = {11.0 12.1 10.1 10.9 13.0 13.0};
net = newlind([P1; P2],[T1; T2],[Pi1; Pi2]);
Y = sim(net,[P1; P2],[Pi1; Pi2]);
Y1 = Y(1,:)
Y2 = Y(2,:)
```

Algorithm

`newlind` calculates weight `W` and bias `B` values for a linear layer from inputs `P` and targets `T` by solving this linear equation in the least squares sense:

```
[W b] * [P; ones] = T
```

See Also

`sim`, `newlin`

Purpose Create a learning vector quantization network

Syntax

```
net = newlvq
net = newlvq(PR,S1,PC,LR,LF)
```

Description Learning vector quantization (LVQ) networks are used to solve classification problems.

`net = newlvq` creates a new network with a dialog box.

`net = newlvq(PR,S1,PC,LR,LF)` takes these inputs,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `S1` - Number of hidden neurons.
- `PC` - `S2` element vector of typical class percentages.
- `LR` - Learning rate, default = 0.01.
- `LF` - Learning function, default = '`learnlv2`'.

returns a new LVQ network.

The learning function LF can be `learnlv1` or `learnlv2`.

Properties `newlvq` creates a two-layer network. The first layer uses the `compet` transfer function, calculates weighted inputs with `negdist`, and net input with `netsum`. The second layer has `purelin` neurons, calculates weighted input with `dotprod` and net inputs with `netsum`. Neither layer has biases.

First layer weights are initialized with `midpoint`. The second layer weights are set so that each output neuron `i` has unit weights coming to it from `PC(i)` percent of the hidden neurons.

Adaption and training are done with `trains` and `trainr`, which both update the first layer weights with the specified learning functions.

Examples The input vectors `P` and target classes `Tc` below define a classification problem to be solved by an LVQ network.

```
P = [-3 -2 -2  0  0  0  0 +2 +2 +3; ...
0 +1  -1  +2 +1  -1  -2 +1  -1   0];
Tc = [1  1  1  2  2  2  2  1  1  1];
```

The target classes Tc are converted to target vectors T. Then, an LVQ network is created (with inputs ranges obtained from P, four hidden neurons, and class percentages of 0.6 and 0.4) and is trained.

```
T = ind2vec(Tc);
net = newlvq(minmax(P),4,[.6 .4]);
net = train(net,P,T);
```

The resulting network can be tested.

```
Y = sim(net,P)
Yc = vec2ind(Y)
```

See Also

sim, init, adapt, train, trains, trainr, learnlv1, learnlv2

newp

Purpose Create a perceptron

Syntax

```
net = newp
net = newp(PR,S,TF,LF)
```

Description Perceptrons are used to solve simple (i.e. linearly separable) classification problems.

`net = newp` creates a new network with a dialog box.

`net = newp(PR,S,TF,LF)` takes these inputs,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `S` - Number of neurons.
- `TF` - Transfer function, default = '`hardlim`'.
- `LF` - Learning function, default = '`learnp`'.

and returns a new perceptron.

The transfer function `TF` can be `hardlim` or `hardlims`. The learning function `LF` can be `learnp` or `learnpn`.

Properties Perceptrons consist of a single layer with the `dotprod` weight function, the `netsum` net input function, and the specified transfer function.

The layer has a weight from the input and a bias.

Weights and biases are initialized with `initzero`.

Adaption and training are done with `trains` and `trainc`, which both update weight and bias values with the specified learning function. Performance is measured with `mae`.

Examples This code creates a perceptron layer with one two-element input (ranges [0 1] and [-2 2]) and one neuron. (Supplying only two arguments to `newp` results in the default perceptron learning function `learnp` being used.)

```
net = newp([0 1; -2 2],1);
```

Here we simulate the network to a sequence of inputs `P`.

```
P1 = {[0; 0] [0; 1] [1; 0] [1; 1]};
Y = sim(net,P1)
```

Here we define a sequence of targets T (together P and T define the operation of an AND gate), and then let the network adapt for 10 passes through the sequence. We then simulate the updated network.

```
T1 = {0 0 0 1};
net.adaptParam.passes = 10;
net = adapt(net,P1,T1);
Y = sim(net,P1)
```

Now we define a new problem, an OR gate, with batch inputs P and targets T.

```
P2 = [0 0 1 1; 0 1 0 1];
T2 = [0 1 1 1];
```

Here we initialize the perceptron (resulting in new random weight and bias values), simulate its output, train for a maximum of 20 epochs, and then simulate it again.

```
net = init(net);
Y = sim(net,P2)
net.trainParam.epochs = 20;
net = train(net,P2,T2);
Y = sim(net,P2)
```

Notes

Perceptrons can classify linearly separable classes in a finite amount of time. If input vectors have a large variance in their lengths, the learnpn can be faster than learnp.

See Also

sim, init, adapt, train, hardlim, hardlims, learnp, learnpn, trains, trainc

Purpose

Design a probabilistic neural network

Syntax

```
net = newpnn
net = newpnn(P,T,spread)
```

Description

Probabilistic neural networks (PNN) are a kind of radial basis network suitable for classification problems.

`net = newpnn` creates a new network with a dialog box.

`net = newpnn(P,T,spread)` takes two or three arguments,

- `P - R x Q` matrix of `Q` input vectors.
- `T - S x Q` matrix of `Q` target class vectors.
- `spread` - Spread of radial basis functions, default = 0.1.

and returns a new probabilistic neural network.

If `spread` is near zero, the network will act as a nearest neighbor classifier. As `spread` becomes larger, the designed network will take into account several nearby design vectors.

Examples

Here a classification problem is defined with a set of inputs `P` and class indices `Tc`.

```
P = [1 2 3 4 5 6 7];
Tc = [1 2 3 2 2 3 1];
```

Here the class indices are converted to target vectors, and a PNN is designed and tested.

```
T = ind2vec(Tc)
net = newpnn(P,T);
Y = sim(net,P)
Yc = vec2ind(Y)
```

Algorithm

`newpnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist`, and its net input with `netprod`. The second layer has `compet` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Only the first layer has biases.

`newpnn` sets the first layer weights to `P'`, and the first layer biases are all set to 0.8326/`spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- `spread`. The second layer weights `W2` are set to `T`.

See Also

`sim`, `ind2vec`, `vec2ind`, `newrb`, `newrbe`, `newgrnn`

References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, pp. 35-55, 1993.

Purpose Design a radial basis network

Syntax

```
net = newrb
[net,tr] = newrb(P,T,goal,spread,MN,DF)
```

Description Radial basis networks can be used to approximate functions. `newrb` adds neurons to the hidden layer of a radial basis network until it meets the specified mean squared error goal.

`net = newrb` creates a new network with a dialog box.

`newrb(P,T,goal,spread,MN, DF)` takes two to these arguments,

- `P - R` x `Q` matrix of `Q` input vectors.
- `T - S` x `Q` matrix of `Q` target class vectors.
- `goal` - Mean squared error goal, default = 0.0.
- `spread` - Spread of radial basis functions, default = 1.0.
- `MN -` Maximum number of neurons, default is Q.
- `DF -` Number of neurons to add between displays, default = 25.

and returns a new radial basis network.

The larger that spread is, the smoother the function approximation will be. Too large a spread means a lot of neurons will be required to fit a fast changing function. Too small a spread means many neurons will be required to fit a smooth function, and the network may not generalize well. Call `newrb` with different spreads to find the best value for a given problem.

Examples Here we design a radial basis network given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

Algorithm `newrb` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist`, and its net input with `netprod`. The

second layer has `purelin` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Both layers have biases.

Initially the `radbas` layer has no neurons. The following steps are repeated until the network's mean squared error falls below `goal`.

1. The network is simulated.
2. The input vector with the greatest error is found.
3. A `radbas` neuron is added with weights equal to that vector.
4. The `purelin` layer weights are redesigned to minimize error.

See Also

`sim`, `newrbe`, `newgrnn`, `newpnn`

Purpose Design an exact radial basis network

Syntax

```
net = newrbe
net = newrbe(P,T,spread)
```

Description Radial basis networks can be used to approximate functions. `newrbe` very quickly designs a radial basis network with zero error on the design vectors.

`net = newrbe` creates a new network with a dialog box.

`newrbe(P,T,spread)` takes two or three arguments,

- `P - R` x `Q` matrix of `Q` input vectors.
- `T - S` x `Q` matrix of `Q` target class vectors.
- `spread` - Spread of radial basis functions, default = 1.0.

and returns a new exact radial basis network.

The larger the spread is, the smoother the function approximation will be. Too large a spread can cause numerical problems.

Examples Here we design a radial basis network given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrbe(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

Algorithm `newrbe` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist`, and its net input with `netprod`. The second layer has `purelin` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Both layers have biases.

`newrbe` sets the first layer weights to `P'`, and the first layer biases are all set to 0.8326/`spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- `spread`.

The second layer weights `IW{2,1}` and biases `b{2}` are found by simulating the first layer outputs `A{1}`, and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

See Also

sim, newrb, newgrnn, newpnn

Purpose

Create a self-organizing map

Syntax

```
net = newsom
net = newsom(PR,[D1,D2,...],TFCN,DFCN,OLR,OSTEPS,TLR,TND)
```

Description

Competitive layers are used to solve classification problems.

`net = newsom` creates a new network with a dialog box.

`net = newsom (PR,[D1,D2,...],TFCN,DFCN,OLR,OSTEPS,TLR,TND)` takes,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `Di` - Size of ith layer dimension, defaults = [5 8].
- `TFCN` - Topology function, default ='`hextop`'.
- `DFCN` - Distance function, default ='`linkdist`'.
- `OLR` - Ordering phase learning rate, default = 0.9.
- `OSTEPS` - Ordering phase steps, default = 1000.
- `TLR` - Tuning phase learning rate, default = 0.02;
- `TND` - Tuning phase neighborhood distance, default = 1.

and returns a new self-organizing map.

The topology function `TFCN` can be `hextop`, `gridtop`, or `randtop`. The distance function can be `linkdist`, `dist`, or `mandist`.

Properties

Self-organizing maps (SOM) consist of a single layer with the `negdist` weight function, `netsum` net input function, and the `compet` transfer function.

The layer has a weight from the input, but no bias. The weight is initialized with `midpoint`.

Adaption and training are done with `trains` and `trainr`, which both update the weight with `learnsom`.

Examples

The input vectors defined below are distributed over an two-dimension input space varying over [0 2] and [0 1]. This data will be used to train a SOM with dimensions [3 5].

```
P = [rand(1,400)*2; rand(1,400)];
net = newsom([0 2; 0 1],[3 5]);
```

```
plotsom(net.layers{1}.positions)
```

Here the SOM is trained and the input vectors are plotted with the map that the SOM's weights have formed.

```
net = train(net,P);
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

See Also

`sim`, `init`, `adapt`, `train`

Purpose Copy matrix or cell array

Syntax `nncopy(X,M,N)`

Description `nncopy(X,M,N)` takes two arguments,

- `X` - `R` x `C` matrix (or cell array).
- `M` - Number of vertical copies.
- `N` - Number of horizontal copies.

and returns a new (`R*M`) x (`C*N`) matrix (or cell array).

Examples

```
x1 = [1 2 3; 4 5 6];
y1 = nncopy(x1,3,2)
x2 = {[1 2]; [3; 4; 5]}
y2 = nncopy(x2,2,3)
```

nnt2c

Purpose

Update NNT 2.0 competitive layer

Syntax

```
net = nnt2c(PR,W,KLR,CLR)
```

Description

`nnt2c(PR,W,KLR,CLR)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W` - `S` x `R` weight matrix.
- `KLR` - Kohonen learning rate, default = 0.01.
- `CLR` - Conscience learning rate, default = 0.001.

and returns a competitive layer.

Once a network has been updated, it can be simulated, initialized, or trained with `sim`, `init`, `adapt`, and `train`.

See Also

`newc`

Purpose

Update NNT 2.0 Elman backpropagation network

Syntax

```
net = nnt2elm(PR,W1,B1,W2,B2,BTF,BLF,PF)
```

Description

`nnt2elm(PR,W1,B1,W2,B2,BTF,BLF,PF)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W1` - `S1` x `(R+S1)` weight matrix.
- `B1` - `S1` x 1 bias vector.
- `W2` - `S2` x `S1` weight matrix.
- `B2` - `S2` x 1 bias vector.
- `BTF` - Backpropagation network training function, default = '`traingdx`'.
- `BLF` - Backpropagation weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns a feed-forward network.

The training function `BTF` can be any of the backpropagation training functions such as `traingd`, `traingdm`, `traingda`, and `traingdx`. Large step-size algorithms, such as `trainlm`, are not recommended for Elman networks.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also

`newelm`

nnt2ff

Purpose

Update NNT 2.0 feed-forward network

Syntax

```
net = nnt2ff(PR,{W1 W2 ...},{B1 B2 ...},{TF1 TF2 ...},BTF,BLR,PF)
```

Description

`nnt2ff(PR,{W1 W2 ...},{B1 B2 ...},{TF1 TF2 ...},BTF,BLR,PF)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `Wi` - Weight matrix for the ith layer.
- `Bi` - Bias vector for the ith layer.
- `TFi` - Transfer function of ith layer, default = '`tansig`'.
- `BTF` - Backpropagation network training function, default = '`traingdx`'.
- `BLF` - Backpropagation weight/bias learning function, default = '`learngdm`'.
- `PF` - Performance function, default = '`mse`'.

and returns a feed-forward network.

The training function `BTF` can be any of the backpropagation training functions such as `traingd`, `traingdm`, `traingda`, `traingdx` or `trainlm`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also

`newff, newcf, newfftd, newelm`

Purpose Update NNT 2.0 Hopfield recurrent network

Syntax

```
net = nnt2p(W,B)
```

Description

`nnt2hop (W,B)` takes these arguments,

- `W` - `S` x `S` weight matrix.
- `B` - `S` x `1` bias vector

and returns a perceptron.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also

`newhop`

nnt2lin

Purpose Update NNT 2.0 linear layer

Syntax

```
net = nnt2lin(PR,W,B,LR)
```

Description

`nnt2lin(PR,W,B)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W` - `S` x `R` weight matrix.
- `B` - `S` x `1` bias vector
- `LR` - Learning rate, default = 0.01;

and returns a linear layer.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also `newlin`

Purpose Update NNT 2.0 learning vector quantization network

Syntax

```
net = nnt2lvq(PR,W1,W2,LR,LF)
```

Description

`nnt2lvq(PR,W1,W2,LR,LF)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W1` - `S1` x `R` weight matrix.
- `W2` - `S2` x `S1` weight matrix.
- `LR` - Learning rate, default = 0.01.
- `LF` - Learning function, default = '`learnlv2`'.

and returns a radial basis network.

The learning function `LF` can be `learnlv1` or `learnlv2`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also `newlvq`

nnt2p

Purpose Update NNT 2.0 perceptron

Syntax

```
net = nnt2p(PR,W,B,TF,LF)
```

Description

`nnt2p(PR,W,B,TF,LF)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W` - `S` x `R` weight matrix.
- `B` - `S` x 1 bias vector.
- `TF` - Transfer function, default = '`hardlim`'.
- `LF` - Learning function, default = '`learnp`'.

and returns a perceptron.

The transfer function `TF` can be `hardlim` or `hardlims`. The learning function `LF` can be `learnp` or `learnpn`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also `newp`

Purpose Update NNT 2.0 radial basis network

Syntax

```
net = nnt2rb(PR,W1,B1,W2,B2)
```

Description

`nnt2rb(PR,W1,B1,W2,B2)` takes these arguments,

- `PR` - `R` x 2 matrix of min and max values for `R` input elements.
- `W1` - `S1` x `R` weight matrix.
- `B1` - `S1` x 1 bias vector.
- `W2` - `S2` x `S1` weight matrix.
- `B2` - `S2` x 1 bias vector.

and returns a radial basis network.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also

`newrb`, `newrbe`, `newgrnn`, `newpnn`

nnt2som

Purpose Update NNT 2.0 self-organizing map

Syntax

```
net = nnt2som(PR,[D1,D2,...],W,OLR,OSTEPS,TLR,TND)
```

Description

`nnt2som(PR,[D1,D2,...],W,OLR,OSTEPS,TLR,TND)` takes these arguments,

`PR` - `R` x 2 matrix of min and max values for `R` input elements.

`Di` - Size of ith layer dimension.

`W` - `S` x `R` weight matrix.

`OLR` - Ordering phase learning rate, default = 0.9.

`OSTEPS` - Ordering phase steps, default = 1000.

`TLR` - Tuning phase learning rate, default = 0.02;

`TND` - Tuning phase neighborhood distance, default = 1.

and returns a self-organizing map.

`nnt2som` assumes that the self-organizing map has a grid topology (`gridtop`) using link distances (`linkdist`). This corresponds with the neighborhood function in NNT 2.0.

The new network will only output 1 for the neuron with the greatest net input. In NNT 2.0 the network would also output 0.5 for that neuron's neighbors.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

See Also `newsom`

Purpose Neural Network Tool - Graphical User Interface

Syntax

```
nntool
```

Description `nntool` opens the Network/Data Manager window, which allows you to import, create, use, and export neural networks and data.

normc

Purpose Normalize the columns of a matrix

Syntax `normc(M)`

Description `normc(M)` normalizes the columns of `M` to a length of 1.

Examples

```
m = [1 2; 3 4];
normc(m)
ans =
     0.3162     0.4472
     0.9487     0.8944
```

See Also `normr`

Purpose Normalized dot product weight function

Syntax

```
Z = normprod(W,P)
df = normprod('deriv')
```

Description `normprod` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`normprod(W,P)` takes these inputs,

`W` - `S` x `R` weight matrix.

`P` - `R` x `Q` matrix of `Q` input (column) vectors.

and returns the `S` x `Q` matrix of normalized dot products.

`normprod('deriv')` returns `''` because `normprod` does not have a derivative function.

Examples Here we define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = normprod(W,P)
```

Network Use You can create a standard network that uses `normprod` by calling `newgrnn`.

To change a network so an input weight uses `normprod`, set `net.inputWeight{i,j}.weightFcn` to '`normprod` . For a layer weight, set `net.inputWeight{i,j}.weightFcn` to '`normprod` .

In either case call `sim` to simulate the network with `normprod`. See `newgrnn` for simulation examples.

Algorithm `normprod` returns the dot product normalized by the sum of the input vector elements.

```
z = w*p/sum(p)
```

See Also `sim, dotprod, negdist, dist`

normr

Purpose Normalize the rows of a matrix

Syntax `normr(M)`

Description `normr(M)` normalizes the columns of `M` to a length of 1.

Examples

```
m = [1 2; 3 4];
normr(m)
ans =
       0.4472     0.8944
       0.6000     0.8000
```

See Also `normc`

Purpose Plot network performance for Bayesian regularization training.

Syntax `plotbr(TR,name,epoch)`

Description `plotbr(tr,name,epoch)` takes these inputs,

`TR` - Training record returned by `train`.

`name` - Training function name, default = ''.

`epoch` - Number of epochs, default = length of training record.

and plots the training sum squared error, the sum squared weight, and the effective number of parameters.

Examples Here are input values `P` and associated targets `T`.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

The code below creates a network and trains it on this problem.

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'trainbr');
[net,tr] = train(net,p,t);
```

During training `plotbr` was called to display the training record. You can also call `plotbr` directly with the final training record `TR`, as shown below.

```
plotbr(tr)
```

plotep

Purpose Plot a weight-bias position on an error surface

Syntax

```
h = plotep(W,B,E)
h = plotep(W,B,E,H)
```

Description `plotep` is used to show network learning on a plot already created by `plotes`.

`plotep(W,B,E)` takes these arguments,

- `W` - Current weight value.
- `B` - Current bias value.
- `E` - Current error.

and returns a vector `H`, containing information for continuing the plot.

`plotep(W,B,E,H)` continues plotting using the vector `H` returned by the last call to `plotep`.

`H` contains handles to dots plotted on the error surface, so they can be deleted next time, as well as points on the error contour, so they can be connected.

See Also `errsurf`, `plotes`

plotes

Purpose

Plot the error surface of a single input neuron

Syntax

`plotes(WV,BV,ES,V)`

Description

`plotes(WV,BV,ES,V)` takes these arguments,

- `WV` - `1` x `N` row vector of values of `W`.
- `BV` - `1` x `M` row vector of values of `B`.
- `ES` - `M` x `N` matrix of error vectors.
- `V` - View, default = [-37.5, 30].

and plots the error surface with a contour underneath.

Calculate the error surface `ES` with `errsurf`.

Examples

```
p = [3 2];
t = [0.4 0.8];
wv = -4:0.4:4; bv = wv;
ES = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```

See Also

`errsurf`

plotpc

Purpose

Plot a classification line on a perceptron vector plot

Syntax

```
plotpc(W,B)
plotpc(W,B,H)
```

Description

`plotpc(W,B)` takes these inputs,

- `W` - `S` x `R` weight matrix (`R` must be 3 or less).
- `B` - `S` x 1 bias vector.

and returns a handle to a plotted classification line.

`plotpc(W,B,H)` takes anadditional input,

- `H` - Handle to last plotted line.

and deletes the last line before plotting the new one.

This function does not change the current axis and is intended to be called after `plotpv`.

Examples

The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```

The following code creates a perceptron with inputs ranging over the values in `P`, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

See Also

`plotpv`

plotperf

Purpose Plot network performance

Syntax

```
plotperf(TR,goal,name,epoch)
```

Description

`plotperf(TR,goal,name,epoch)` takes these inputs,

`TR` - Training record returned by `train`.

`goal` - Performance goal, default = `NaN`.

`name` - Training function name, default = `''`.

`epoch` - Number of epochs, default = length of training record.

and plots the training performance, and if available, the performance goal, validation performance, and test performance.

Examples

Here are eight input values `P` and associated targets `T`, plus a like number of validation inputs `VV.P` and targets `VV.T`.

```
P = 1:8; T = sin(P);
VV.P = P; VV.T = T+rand(1,8)*0.1;
```

The code below creates a network and trains it on this problem.

```
net = newff(minmax(P),[4 1],{'tansig','tansig'});
[net,tr] = train(net,P,T,[],[],VV);
```

During training `plotperf` was called to display the training record. You can also call `plotperf` directly with the final training record `TR`, as shown below.

```
plotperf(tr)
```

plotpv

Purpose

Plot perceptron input/target vectors

Syntax

```
plotpv(P,T)
plotpv(P,T,V)
```

Description

`plotpv(P,T)` take these inputs,

- `P` - `R x Q` matrix of input vectors (`R` must be 3 or less).
- `T` - `S` x `Q` matrix of binary target vectors (`S` must be 3 or less).

and plots column vectors in `P` with markers based on `T`.

`plotpv(P,T,V)` takes an additional input,

- `V` - Graph limits = `[x_min x_max y_min y_max]`

and plots the column vectors with limits set by `V`.

Examples

The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```

The following code creates a perceptron with inputs ranging over the values in `P`, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

See Also

`plotpc`

Purpose

Plot self-organizing map

Syntax

`plotsom(pos)`

`plotsom(W,D,ND)`

Description

`plotsom(pos)` takes one argument,

`POS - NxS` matrix of `S N`-dimension neural positions and plots the neuron positions with red dots, linking the neurons within a Euclidean distance of 1.

`plotsom(W,d,nd)` takes three arguments,

- `W` - `SxR` weight matrix.
- `D` - `SxS` distance matrix.
- `ND` - Neighborhood distance, default = 1.

and plots the neuron's weight vectors with connections between weight vectors whose neurons are within a distance of 1.

Examples

Here are some neat plots of various layer topologies:

```
pos = hextop(5,6); plotsom(pos)
pos = gridtop(4,5); plotsom(pos)
pos = randtop(18,12); plotsom(pos)
pos = gridtop(4,5,2); plotsom(pos)
pos = hextop(4,4,3); plotsom(pos)
```

See `newsom` for an example of plotting a layer's weight vectors with the input vectors they map.

See Also

`newsom, learnsom, initsom.`

plotv

Purpose Plot vectors as lines from the origin

Syntax

```
plotv(M,T)
```

Description `plotv(M,T)` takes two inputs,

`M` - `R` x `Q` matrix of `Q` column vectors with `R` elements.

`T` - (optional) the line plotting type, default = `'-'`.

and plots the column vectors of `M`.

`R` must be 2 or greater. If `R` is greater than two, only the first two rows of `M` are used for the plot.

Examples

```
plotv([-.4 0.7 .2; -0.5 .1 0.5],'-')
```

Purpose Plot vectors with different colors

Syntax `plotvec(X,C,M)`

Description `plotvec(X,C,M)` takes these inputs,

- `X` - Matrix of (column) vectors.
- `C` - Row vector of color coordinate.
- `M` - Marker, default = `'+'`.

and plots each ith vector in `X` with a marker `M` and using the ith value in `C` as the color coordinate.

`plotvec(X)` only takes a matrix `X` and plots each ith vector in `X` with marker `'+'` using the index `i` as the color coordinate.

Examples

```
x = [0 1 0.5 0.7; -1 2 0.5 0.1];
c = [1 2 3 4];
plotvec(x,c)
```

pnormc

Purpose Pseudo-normalize columns of a matrix

Syntax `pnormc(X,R)`

Description `pnormc(X,R)` takes these arguments,

`X` - `M` x `N` matrix.

`R` - (optional) radius to normalize columns to, default = 1.

and returns `X` with an additional row of elements, which results in new column vector lengths of `R`.

Caution: For this function to work properly, the columns of `X` must originally have vector lengths less than `R`.

Examples

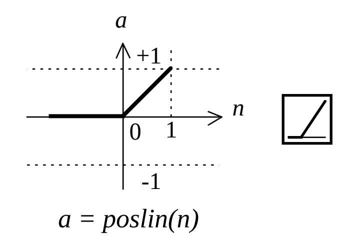
```
x = [0.1 0.6; 0.3 0.1];
y = pnormc(x)
```

See Also `normc`, `normr`

Purpose

Positive linear transfer function

Graph and Symbol



Positive Linear Transfer Funct.

Syntax

```
A = poslin(N)
info = poslin(code)
```

Description

`poslin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`poslin(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns the maximum of 0 and each element of `N`.

`poslin(code)` returns useful information for each `code` string:

`'deriv'` - Name of derivative function.

`'name'` - Full name.

`'output'` - Output range.

`'active'` - Active input range.

Examples

Here is the code to create a plot of the `poslin` transfer function.

```
n = -5:0.1:5;
a = poslin(n);
plot(n,a)
```

Network Use

To change a network so that a layer uses `poslin`, set `net.layers{i}.transferFcn` to '`poslin`'.

Call `sim` to simulate the network with `poslin`.

Algorithm

The transfer function `poslin` returns the output `n` if `n` is greater than or equal to zero and 0 if `n` is less than or equal to zero.

```
poslin(n) = n, if n >= 0; = 0, if n <= 0.
```

See Also

`sim`, `purelin`, `satlin`, `satlins`

Purpose Postprocess data that has been preprocessed by `premnmx`

Syntax

```
[P,T] = postmnmx(PN,minp,maxp,TN,mint,maxt)
[p] = postmnmx(PN,minp,maxp)
```

Description

`postmnmx` postprocesses the network training set that was preprocessed by `premnmx`. It converts the data back into unnormalized units.

`postmnmx` takes these inputs,

- `PN` - `R` x `Q` matrix of normalized input vectors.
- `minp`- `R` x 1 vector containing minimums for each `P`.
- `maxp`- `R` x 1 vector containing maximums for each `P`.
- `TN` - `S` x `Q` matrix of normalized target vectors.
- `mint`- `S` x 1 vector containing minimums for each `T`.
- `maxt`- `S` x 1 vector containing maximums for each `T`.

and returns,

- `P` - `R` x `Q` matrix of input (column) vectors.
- `T` - `R` x `Q` matrix of target vectors.

Examples

In this example we normalize a set of training data with `premnmx`, create and train a network using the normalized data, simulate the network, unnormalize the output of the network using `postmnmx`, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
an = sim(net,pn);
[a] = postmnmx(an,mint,maxt);
[m,b,r] = postreg(a,t);
```

Algorithm

```
p = 0.5(pn+1)*(maxp-minp) + minp;
```

See Also `premnmx`, `prepca`, `poststd`

postreg

Purpose Postprocess the trained network response with a linear regression

Syntax

```
[M,B,R] = postreg(A,T)
```

Description

`postreg` postprocesses the network training set by performing a linear regression between each element of the network response and the corresponding target.

`postreg(A,T)` takes these inputs,

- `A` - `1 x Q` array of network outputs. One element of the network output.
- `T` - `1 x Q` array of targets. One element of the target vector.

and returns,

- `M` - Slope of the linear regression.
- `B` - `Y` intercept of the linear regression.
- `R` - Regression `R`-value. `R=1` means perfect correlation.

Examples

In this example we normalize a set of training data with `prestd`, perform a principal component transformation on the normalized data, create and train a network using the `pca` data, simulate the network, unnormalize the output of the network using `poststd`, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.02);
net = newff(minmax(ptrans),[5 1],{'tansig''purelin'},'trainlm');
net = train(net,ptrans,tn);
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
[m,b,r] = postreg(a,t);
```

Algorithm

Performs a linear regression between the network response and the target, and then computes the correlation coefficient (`R`-value) between the network response and the target.

See Also

`premnmx`, `prepca`

poststd

Purpose Postprocess data which has been preprocessed by `prestd`

Syntax

```
[P,T] = poststd(PN,meanp,stdp,TN,meant,stdt)
[p] = poststd(PN,meanp,stdp)
```

Description `poststd` postprocesses the network training set that was preprocessed by `prestd`. It converts the data back into unnormalized units.

`poststd` takes these inputs,

- `PN` - `R` x `Q` matrix of normalized input vectors.
- `meanp` - `R` x 1 vector containing standard deviations for each `P`.
- `stdp` - `R` x 1 vector containing standard deviations for each `P`.
- `TN` - `S` x `Q` matrix of normalized target vectors.
- `meant` - `S x 1` vector containing standard deviations for each `T`.
- `stdt` - `S x 1` vector containing standard deviations for each `T`.

and returns,

- `P` - `R` x `Q` matrix of input (column) vectors.
- `T` - `S` x `Q` matrix of target vectors.

Examples In this example we normalize a set of training data with `prestd`, create and train a network using the normalized data, simulate the network, unnormalize the output of the network using `poststd`, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
an = sim(net,pn);
a = poststd(an,meant,stdt);
[m,b,r] = postreg(a,t);
```

Algorithm

```
p = stdp*pn + meanp;
```

See Also `premnmx, prepca, postmnmx, prestd`

premnmx

Purpose Preprocess data so that minimum is -1 and maximum is 1

Syntax

```
[PN,minp,maxp,TN,mint,maxt] = premnmx(P,T)
[PN,minp,maxp] = premnmx(P)
```

Description `premnmx` preprocesses the network training set by normalizing the inputs and targets so that they fall in the interval `[-1,1]`.

`premnmx(P,T)` takes these inputs,

- `P` - `R` x `Q` matrix of input (column) vectors.
- `T` - `S` x `Q` matrix of target vectors.

and returns,

- `PN` - `R` x `Q` matrix of normalized input vectors.
- `minp` - `R` x `1` vector containing minimums for each `P`.
- `maxp` - `R` x `1` vector containing maximums for each `P`.
- `TN` - `S` x `Q` matrix of normalized target vectors.
- `mint` - `S` x `1` vector containing minimums for each `T`.
- `maxt` - `S` x `1` vector containing maximums for each `T`.

Examples Here is the code to normalize a given data set so that the inputs and targets will fall in the range `[-1,1]`.

```
p = [-10 -7.5 -5 -2.5 0 2.5 5 7.5 10];
t = [0 7.07 -10 -7.07 0 7.07 10 7.07 0];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
```

If you just want to normalize the input,

```
[pn,minp,maxp] = premnmx(p);
```

Algorithm

```
pn = 2*(p-minp)/(maxp-minp) - 1;
```

See Also `prestd`, `prepca`, `postmnmx`

Purpose Principal component analysis

Syntax

```
[ptrans,transMat] = prepca(P,min_frac)
```

Description

prepca preprocesses the network input training set by applying a principal component analysis. This analysis transforms the input data so that the elements of the input vector set will be uncorrelated. In addition, the size of the input vectors may be reduced by retaining only those components which contribute more than a specified fraction (min_frac) of the total variation in the data set.

prepca(P,min_frac) takes these inputs

P - R x Q matrix of centered input (column) vectors.

min_frac - Minimum fraction variance component to keep.

and returns

ptrans - Transformed data set.

transMat - Transformation matrix.

Examples

Here is the code to perform a principal component analysis and retain only those components that contribute more than two percent to the variance in the data set. prestd is called first to create zero mean data, which is needed for prepca.

```
p=[-1.5 -0.58 0.21 -0.96 -0.79; -2.2 -0.87 0.31 -1.4  -1.2];
[pn,meanp,stdp] = prestd(p);
[ptrans,transMat] = prepca(pn,0.02);
```

Since the second row of p is almost a multiple of the first row, this example will produce a transformed data set that contains only one row.

Algorithm

This routine uses singular value decomposition to compute the principal components. The input vectors are multiplied by a matrix whose rows consist of the eigenvectors of the input covariance matrix. This produces transformed input vectors whose components are uncorrelated and ordered according to the magnitude of their variance.

Those components that contribute only a small amount to the total variance in the data set are eliminated. It is assumed that the input data set has already been normalized so that it has a zero mean. The function `prestd` can be used to normalize the data.

See Also

`prestd`, `premnmx`

References

Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

Purpose Preprocess data so that its mean is 0 and the standard deviation is 1

Syntax

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t)
[pn,meanp,stdp] = prestd(p)
```

Description `prestd` preprocesses the network training set by normalizing the inputs and targets so that they have means of zero and standard deviations of 1.

`prestd(p,t)` takes these inputs,

- `p` - `R` x `Q` matrix of input (column) vectors.
- `t` - `S` x `Q` matrix of target vectors.

and returns,

- `pn` - `R` x `Q` matrix of normalized input vectors.
- `meanp` - `R` x 1 vector containing mean for each `P`.
- `stdp` - `R` x 1 vector containing standard deviations for each `P`.
- `tn` - `S` x `Q` matrix of normalized target vectors.
- `meant` - `S` x 1 vector containing mean for each `T`.
- `stdt` - `S` x 1 vector containing standard deviations for each `T`.

Examples Here is the code to normalize a given data set so that the inputs and targets will have means of zero and standard deviations of 1.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
```

If you just want to normalize the input,

```
[pn,meanp,stdp] = prestd(p);
```

Algorithm

```
pn = (p-meanp)/stdp;
```

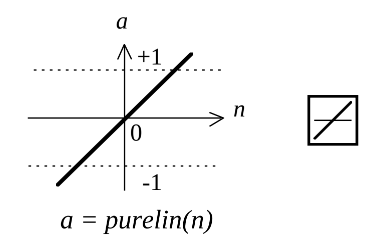
See Also `premnmx`, `prepca`

purelin

Purpose

Linear transfer function

Graph and Symbol



Linear Transfer Function

Syntax

```
A = purelin(N)
info = purelin(code)
```

Description

`purelin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`purelin(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns `N`.

`purelin(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here is the code to create a plot of the `purelin` transfer function.

```
n = -5:0.1:5;
a = purelin(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `purelin` by calling `newlin` or `newlind`.

To change a network so a layer uses `purelin`, set `net.layers{i}.transferFcn` to `'purelin'`.

In either case, call `sim` to simulate the network with `purelin`. See `newlin` or `newlind` for simulation examples.

Algorithm

```
purelin(n) = n
```

See Also

`sim`, `dpurelin`, `satlin`, `satlins`

quant

Purpose

Discretize values as multiples of a quantity

Syntax

`quant(X,Q)`

Description

`quant(X,Q)` takes two inputs,

`X` - Matrix, vector or scalar.

`Q` - Minimum value.

and returns values in `X` rounded to nearest multiple of `Q`.

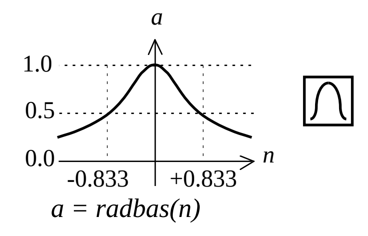
Examples

```
x = [1.333 4.756 -3.897];
y = quant(x,0.1)
```

Purpose

Radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

```
A = radbas(N)
info = radbas(code)
```

Description

`radbas` is a transfer function. Transfer functions calculate a layer's output from its net input.

`radbas(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns each element of `N` passed through a radial basis function.

`radbas(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here we create a plot of the `radbas` transfer function.

```
n = -5:0.1:5;
a = radbas(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `radbas` by calling `newpnn` or `newgrnn`.

To change a network so that a layer uses `radbas`, set `net.layers{i}.transferFcn` to '`radbas`'.

In either case, call `sim` to simulate the network with `radbas`. See `newpnn` or `newgrnn` for simulation examples.

Algorithm

`radbas(N)` calculates its output as:

```
a = exp(-n2)
```

See Also

`sim, tribas, dradbas`

Purpose Normalized column weight initialization function

Syntax

```
W = randnc(S,PR)
W = randnc(S,R)
```

Description

randnc is a weight initialization function.

randnc(S,P) takes two inputs,

S - Number of rows (neurons).

PR - R x 2 matrix of input value ranges = [Pmin Pmax].

and returns an S x R random matrix with normalized columns.

Can also be called as randnc(S,R).

Examples

A random matrix of four normalized three-element columns is generated:

```
M = randnc(3,4)
M =
     0.6007    0.4715    0.2724    0.5596
     0.7628    0.6967    0.9172    0.7819
     0.2395    0.5406    0.2907    0.2747
```

See Also

randnr

randnr

Purpose Normalized row weight initialization function

Syntax

```
W = randnr(S,PR)
W = randnr(S,R)
```

Description

`randnr` is a weight initialization function.

`randnr(S,PR)` takes two inputs,

`S` - Number of rows (neurons).

`PR` - `R` x 2 matrix of input value ranges = `[Pmin Pmax]`.

and returns an `S` x `R` random matrix with normalized rows.

Can also be called as `randnr(S,R)`.

Examples

A matrix of three normalized four-element rows is generated:

```
M = randnr(3,4)
M =
    0.9713    0.0800    0.1838    0.1282
    0.8228    0.0338    0.1797    0.5381
    0.3042    0.5725    0.5436    0.5331
```

See Also

`randnc`

Purpose

Symmetric random weight/bias initialization function

Syntax

```
W = rands(S,PR)
M = rands(S,R)
v = rands(S);
```

Description

`rands` is a weight/bias initialization function.

`rands(S,PR)` takes,

- `S` - Number of neurons.
- `PR` - `R` x 2 matrix of `R` input ranges.

and returns an `S`-by-`R` weight matrix of random values between -1 and 1.

`rands(S,R)` returns an `S`-by-`R` matrix of random values. `rands(S)` returns an `S`-by-1 vector of random values.

Examples

Here three sets of random values are generated with `rands`.

```
rands(4,[0 1; -2 2])
rands(4)
rands(2,3)
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `rands`:

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set each `net.inputWeights{i,j}.initFcn` to '`rands`'. Set each `net.layerWeights{i,j}.initFcn` to '`rands`'. Set each `net.biases{i}.initFcn` to '`rands`'.

To initialize the network call `init`.

See Also

`randnr, randnc, initwb, initlay, init`

randtop

Purpose Random layer topology function

Syntax `pos = randtop(dim1,dim2,...,dimN)`

Description `randtop` calculates the neuron positions for layers whose neurons are arranged in an `N` dimensional random pattern.

`randtop(dim1,dim2,...,dimN))` takes `N` arguments,

`dimi` - Length of layer in dimension `i`.

and returns an `N` x `S` matrix of `N` coordinate vectors, where `S` is the product of `dim1*dim2*...*dimN`.

Examples This code creates and displays a two-dimensional layer with 192 neurons arranged in a 16-by-12 random pattern.

```
pos = randtop(16,12); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so that the layer is very unorganized, as is evident in the plot.

```
W = rands(192,2); plotsom(W,dist(pos))
```

See Also `gridtop`, `hextop`

Purpose

Change network weights and biases to previous initialization values

Syntax

```
net = revert(net)
```

Description

`revert (net)` returns neural network `net` with weight and bias values restored to the values generated the last time the network was initialized.

If the network has been altered so that it has different weight and bias connections or different input or layer sizes, then `revert` cannot set the weights and biases to their previous values and they will be set to zeros instead.

Examples

Here a perceptron is created with a two-element input (with ranges of 0 to 1, and -2 to 2) and one neuron. Once it is created we can display the neuron's weights and bias.

```
net = newp([0 1;-2 2],1);
```

The initial network has weights and biases with zero values.

```
net.iw{1,1}, net.b{1}
```

We can change these values as follows.

```
net.iw{1,1} = [1 2];
net.b{1} = 5;
net.iw{1,1}, net.b{1}
```

We can recover the network's initial values as follows.

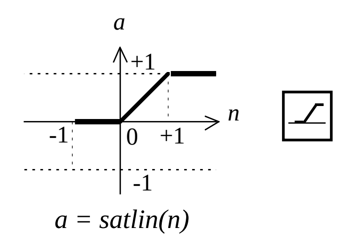
```
net = revert(net);
net.iw{1,1}, net.b{1}
```

See Also

`init`, `sim`, `adapt`, `train`.

Purpose Saturating linear transfer function

Graph and Symbol



Satlin Transfer Function

Syntax

```
A = satlin(N)
info = satlin(code)
```

Description

`satlin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`satlin(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors,

and returns values of `N` truncated into the interval `[-1, 1]`.

`satlin(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here is the code to create a plot of the `satlin` transfer function.

```
n = -5:0.1:5;
a = satlin(n);
plot(n,a)
```

Network Use

To change a network so that a layer uses `satlin`, set `net.layers{i}.transferFcn` to '`satlin`'.

Call `sim` to simulate the network with `satlin`. See `newhop` for simulation examples.

Algorithm

```
satlin(n) = 0, if n <= 0; n, if 0 <= n <= 1; 1, if 1 <= n.
```

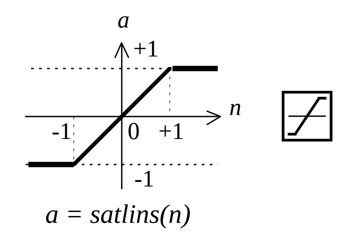
See Also

`sim`, `poslin`, `satlins`, `purelin`

Purpose

Symmetric saturating linear transfer function

Graph and Symbol



Satlins Transfer Function

Syntax

```
A = satlins(N)
info = satlins(code)
```

Description

`satlins` is a transfer function. Transfer functions calculate a layer's output from its net input.

`satlins(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns values of `N` truncated into the interval `[-1, 1]`.

`satlins(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here is the code to create a plot of the `satlins` transfer function.

```
n = -5:0.1:5;
a = satlins(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `satlins` by calling `newhop`.

To change a network so that a layer uses `satlins`, set `net.layers{i}.transferFcn` to `'satlins'`.

In either case, call `sim` to simulate the network with `satlins`. See `newhop` for simulation examples.

Algorithm

```
satlins(n) = -1, if n <= -1; n, if -1 <= n <= 1; 1, if 1 <= n.
```

See Also

`sim, satlin, poslin, purelin`

seq2con

Purpose Convert sequential vectors to concurrent vectors

Syntax

```
b = seq2con(s)
```

Description The Neural Network Toolbox represents batches of vectors with a matrix, and sequences of vectors with multiple columns of a cell array.

`seq2con` and `con2seq` allow concurrent vectors to be converted to sequential vectors, and back again.

`seq2con(S)` takes one input,

`S - N` x `TS` cell array of matrices with `M` columns.

and returns,

`B - N` x `1` cell array of matrices with `M*TS` columns.

Examples Here three sequential values are converted to concurrent values.

```
p1 = {1 4 2}
p2 = seq2con(p1)
```

Here two sequences of vectors over three time steps are converted to concurrent vectors.

```
p1 = {[1; 1] [5; 4] [1; 2]; [3; 9] [4; 1] [9; 8]}
p2 = seq2con(p1)
```

See Also `con2seq`, `concur`

Purpose

Set all network weight and bias values with a single vector

Syntax

```
net = setx(net,X)
```

Description

This function sets a networks weight and biases to a vector of values.

```
net = setx(net,X)
```

- `net` - Neural network.
- `X` - Vector of weight and bias values.

Examples

Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

The network has six weights (3 neurons * 2 input elements) and three biases (3 neurons) for a total of nine weight and bias values. We can set them to random values as follows:

```
net = setx(net,rand(9,1));
```

We can then view the weight and bias values as follows:

```
net.iw{1,1}
net.b{1}
```

See Also

`getx`, `formx`

sim

Purpose Simulate a neural network

Syntax

```
[Y,Pf,Af,E,perf] = sim(net,P,Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,{Q TS},Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,Q,Pi,Ai,T)
```

To Get Help Type `help network/sim`

Description

`sim` simulates neural networks.

`[Y,Pf,Af,E,perf] = sim(net,P,Pi,Ai,T)` takes,

- `net` - Network.
- `P` - Network inputs.
- `Pi` - Initial input delay conditions, default = zeros.
- `Ai` - Initial layer delay conditions, default = zeros.
- `T` - Network targets, default = zeros.

and returns,

- `Y` - Network outputs.
- `Pf` - Final input delay conditions.
- `Af` - Final layer delay conditions.
- `E` - Network errors.
- `perf` - Network performance.

Note that arguments `Pi`, `Ai`, `Pf`, and `Af` are optional and need only be used for networks that have input or layer delays.

`sim`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

```
P  - Ni x TS cell array, each element P{i,ts} is an Ri x Q matrix.
Pi - Ni x ID cell array, each element Pi{i,k} is an Ri x Q matrix.
Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.
T  - Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix.
Y  - NO x TS cell array, each element Y{i,ts} is a Ui x Q matrix.
Pf - Ni x ID cell array, each element Pf{i,k} is an Ri x Q matrix.
Af - Nl x LD cell array, each element Af{i,k} is an Si x Q matrix.
E  - Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix.
```

where

```
Ni = net.numInputs
Nl = net.numLayers
No = net.numOutputs
D  = net.numInputDelays
LD = net.numLayerDelays
TS = Number of time steps
Q  = Batch size
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Ui = net.outputs{i}.size
```

The columns of `Pi`, `Ai`, `Pf`, and `Af` are ordered from oldest delay condition to most recent:

```
Pi{i,k} = input i at time ts=k-ID.
Pf{i,k} = input i at time ts=TS+k-ID.
Ai{i,k} = layer output i at time ts=k-LD.
Af{i,k} = layer output i at time ts=TS+k-LD.
```

The matrix format can be used if only one time step is to be simulated (`TS = 1`). It is convenient for networks with only one input and output, but can also be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument into a single matrix:

`P  - (sum of Ri) x Q` matrix

`Pi - (sum of Ri) x (ID*Q)` matrix.

`Ai - (sum of Si) x (LD*Q)` matrix.

`T  - (sum of Vi)xQ` matrix

`Y  - (sum of Ui) x Q` matrix.

`Pf - (sum of Ri) x (ID*Q)` matrix.

`Af - (sum of Si) x (LD*Q)` matrix.

`E  - (sum of Vi)xQ` matrix.

`[Y,Pf,Af] = sim(net,{Q TS},Pi,Ai)` is used for networks which do not have an input, such as Hopfield networks, when cell array notation is used.

Examples

Here `newp` is used to create a perceptron layer with a two-element input (with ranges of `[0 1]`), and a single neuron.

```
net = newp([0 1;0 1],1);
```

Here the perceptron is simulated for an individual vector, a batch of three vectors, and a sequence of three vectors.

```
p1 = [.2; .9]; a1 = sim(net,p1)
p2 = [.2 .5 .1; .9 .3 .7]; a2 = sim(net,p2)
p3 = {[.2; .9] [.5; .3] [.1; .7]}; a3 = sim(net,p3)
```

Here `newlind` is used to create a linear layer with a three-element input, two neurons.

```
net = newlin([0 2;0 2;0 2],2,[0 1]);
```

Here the linear layer is simulated with a sequence of two input vectors using the default initial input delay conditions (all zeros).

```
p1 = {[2; 0.5; 1] [1; 1.2; 0.1]};
[y1,pf] = sim(net,p1)
```

Here the layer is simulated for three more vectors using the previous final input delay conditions as the new initial delay conditions.

```
p2 = {[0.5; 0.6; 1.8] [1.3; 1.6; 1.1] [0.2; 0.1; 0]};
```

```
[y2,pf] = sim(net,p2,pf)
```

Here newelm is used to create an Elman network with a one-element input, and a layer 1 with three tansig neurons followed by a layer 2 with two purelin neurons. Because it is an Elman network it has a tap delay line with a delay of 1 going from layer 1 to layer 1.

```
net = newelm([0 1],[3 2],{'tansig','purelin'});
```

Here the Elman network is simulated for a sequence of three values using default initial delay conditions.

```
p1 = {0.2 0.7 0.1};
[y1,pf,af] = sim(net,p1)
```

Here the network is simulated for four more values, using the previous final delay conditions as the new initial delay conditions.

```
p2 = {0.1 0.9 0.8 0.4};
[y2,pf,af] = sim(net,p2,pf,af)
```

Algorithm

sim uses these properties to simulate a network net.

```
net.numInputs, net.numLayers
net.outputConnect, net.biasConnect
net.inputConnect, net.layerConnect
```

These properties determine the network's weight and bias values, and the number of delays associated with each weight:

```
net.IW{i,j}
net.LW{i,j}
net.b{i}
net.inputWeights{i,j}.delays
net.layerWeights{i,j}.delays
```

These function properties indicate how sim applies weight and bias values to inputs to get each layer's output:

```
net.inputWeights{i,j}.weightFcn
net.layerWeights{i,j}.weightFcn
net.layers{i}.netInputFcn
net.layers{i}.transferFcn
```

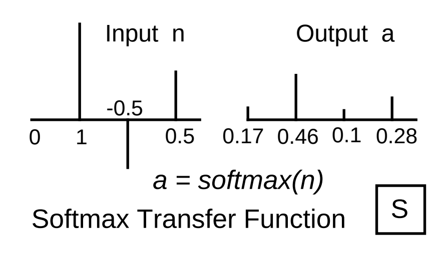
See Chapter 2 for more information on network simulation.

See Also `init`, `adapt`, `train`, `revert`

Purpose

Soft max transfer function

Graph and Symbol



Syntax

```
A = softmax(N)
```

```
info = softmax(code)
```

Description

`softmax` is a transfer function. Transfer functions calculate a layer's output from its net input.

`softmax(N)` takes one input argument,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns output vectors with elements between 0 and 1, but with their size relations intact.

`softmax('code')` returns information about this function.

These codes are defined:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

`compet` does not have a derivative function.

Examples

Here we define a net input vector `N`, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = softmax(n);
subplot(2,1,1), bar(n), ylabel('n')
```

```
subplot(2,1,2), bar(a), ylabel('a')
```

Network Use

To change a network so that a layer uses `softmax`, set `net.layers{i,j}.transferFcn` to '`softmax`'.

Call `sim` to simulate the network with `softmax`. See `newc` or `newpnn` for simulation examples.

See Also

`sim`, `compet`

Purpose

One-dimensional minimization using backtracking

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)
```

Description

`srchbac` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called backtracking.

`srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)` takes these inputs,

- `net` - Neural network.
- `X` - Vector containing current values of weights and biases.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `dX` - Search direction vector.
- `gX` - Gradient vector.
- `perf` - Performance value at current `X`.
- `dperf` - Slope of performance value at current `X` in direction of `dX`.
- `delta` - Initial step size.
- `tol` - Tolerance on search.
- `ch_perf` - Change in performance on previous step.

and returns,

- `a` - Step size, which minimizes performance.
- `gX` - Gradient at new minimum point.
- `perf` - Performance value at new minimum point.
- `retcode` - Return code which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` - New initial step size. Based on the current step size.

`tol` - New tolerance on search.

Parameters used for the backstepping algorithm are:

`alpha` - Scale factor that determines sufficient reduction in `perf`.

`beta` - Scale factor that determines sufficiently large step size.

`low_lim` - Lower limit on change in step size.

`up_lim` - Upper limit on change in step size.

`maxstep` - Maximum step length.

`minstep` - Minimum step length.

`scale_tol` - Parameter which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchbac` search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbac';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchbac` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchbac`:

1. Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam.searchFcn` to '`srchbac`'.

The `srchbac` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Algorithm

`srchbac` locates the minimum of the performance function in the search direction `dX`, using the backtracking algorithm described on page 126 and 328 of Dennis and Schnabel's book noted below.

See Also

`srchcha`, `srchgol`, `srchhyb`

References

Dennis, J. E., and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

Purpose One-dimensional interval location using Brent's method

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchbre` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called Brent's technique.

`srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

- `net` - Neural network.
- `X` - Vector containing current values of weights and biases.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `dX` - Search direction vector.
- `gX` - Gradient vector.
- `perf` - Performance value at current `X`.
- `dperf` - Slope of performance value at current `X` in direction of `dX`.
- `delta` - Initial step size.
- `tol` - Tolerance on search.
- `ch_perf` - Change in performance on previous step.

and returns,

- `a` - Step size, which minimizes performance.
- `gX` - Gradient at new minimum point.
- `perf` - Performance value at new minimum point.
- `retcode` - Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` - New initial step size. Based on the current step size.

`tol` - New tolerance on search.

Parameters used for the brent algorithm are:

`alpha` - Scale factor, which determines sufficient reduction in `perf`.

`beta` - Scale factor, which determines sufficiently large step size.

`bmax` - Largest step size.

`scale_tol` - Parameter which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchbac` search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbre';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchbre` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchbre`:

1. Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam.searchFcn` to '`srchbre`'.

The `srchbre` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Algorithm

`srchbre` brackets the minimum of the performance function in the search direction `dX`, using Brent's algorithm described on page 46 of Scales (see reference below). It is a hybrid algorithm based on the golden section search and the quadratic approximation.

See Also

`srchbac, srchcha, srchgol, srchhyb`

References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

srchcha

Purpose One-dimensional minimization using Charalambous' method

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchcha` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique based on Charalambous' method.

`srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

- `net` - Neural network.
- `X` - Vector containing current values of weights and biases.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `dX` - Search direction vector.
- `gX` - Gradient vector.
- `perf` - Performance value at current `X`.
- `dperf` - Slope of performance value at current `X` in direction of `dX`.
- `delta` - Initial step size.
- `tol` - Tolerance on search.
- `ch_perf` - Change in performance on previous step.

and returns,

- `a` - Step size, which minimizes performance.
- `gX` - Gradient at new minimum point.
- `perf` - Performance value at new minimum point.
- `retcode` - Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` - New initial step size. Based on the current step size.

`tol` - New tolerance on search.

Parameters used for the Charalambous algorithm are:

`alpha` - Scale factor, which determines sufficient reduction in `perf`.

`beta` - Scale factor, which determines sufficiently large step size.

`gama` - Parameter to avoid small reductions in performance. Usually set to 0.1.

`scale_tol` - Parameter, which relates the tolerance `tol` to the initial step size delta. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
```

```
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchcha search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchcha';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses srchcha with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf, using the line search function srchcha:

1. Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2. Set net.trainParam.searchFcn to 'srchcha'.

The srchcha function can be used with any of the following training functions: traincgf, traincgb, traincgp, trainbfg, trainoss.

Algorithm

srchcha locates the minimum of the performance function in the search direction dX, using an algorithm based on the method described in Charalambous (see reference below).

See Also

srchbac, srchbre, srchgol, srchhyb

References

Charalambous, C.,“Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, June 1992.

srchgol

Purpose

One-dimensional minimization using golden section search

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchgol` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called the golden section search.

`srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

- `net` - Neural network.
- `X` - Vector containing current values of weights and biases.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `dX` - Search direction vector.
- `gX` - Gradient vector.
- `perf` - Performance value at current `X`.
- `dperf` - Slope of performance value at current `X` in direction of `dX`.
- `delta` - Initial step size.
- `tol` - Tolerance on search.
- `ch_perf` - Change in performance on previous step.

and returns,

- `a` - Step size, which minimizes performance.
- `gX` - Gradient at new minimum point.
- `perf` - Performance value at new minimum point.
- `retcode` - Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` - New initial step size. Based on the current step size.

`tol` - New tolerance on search.

Parameters used for the golden section algorithm are:

`alpha` - Scale factor, which determines sufficient reduction in `perf`.

`bmax` - Largest step size.

`scale_tol` - Parameter, which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf, traincgb, traincgp, trainbfg, trainoss.`

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchgol` search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchgol';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchgol` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchgol`:

1. Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam.searchFcn` to '`srchgol`'.

The `srchgol` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Algorithm

`srchgol` locates the minimum of the performance function in the search direction `dX`, using the golden section search. It is based on the algorithm as described on page 33 of Scales (see reference below).

See Also

`srchbac, srchbre, srchcha, srchhyb`

References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

Purpose One-dimensional minimization using a hybrid bisection-cubic search

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchhyb(net,X,P,T,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

srchhyb is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique that is a combination of a bisection and a cubic interpolation.

srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf) takes these inputs,

net - Neural network.

X - Vector containing current values of weights and biases.

Pd - Delayed input vectors.

Tl - Layer target vectors.

Ai - Initial input delay conditions.

Q - Batch size.

TS - Time steps.

dX - Search direction vector.

gX - Gradient vector.

perf - Performance value at current X.

dperf - Slope of performance value at current X in direction of dX.

delta - Initial step size.

tol - Tolerance on search.

ch_perf - Change in performance on previous step.

and returns,

a - Step size, which minimizes performance.

gX - Gradient at new minimum point.

perf - Performance value at new minimum point.

retcode - Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` - New initial step size. Based on the current step size.

`tol` - New tolerance on search.

Parameters used for the hybrid bisection-cubic algorithm are:

`alpha` - Scale factor, which determines sufficient reduction in `perf`.

`beta` - Scale factor, which determines sufficiently large step size.

`bmax` - Largest step size.

`scale_tol` - Parameter, which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
```

```
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchhyb search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchhyb';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses srchhyb with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf, using the line search function srchhyb:

1 Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2 Set net.trainParam.searchFcn to 'srchhyb'.

The srchhyb function can be used with any of the following training functions: traincgf, traincgb, traincgp, trainbfg, trainoss.

Algorithm

srchhyb locates the minimum of the performance function in the search direction dX, using the hybrid bisection-cubic interpolation algorithm described on page 50 of Scales (see reference below).

See Also

srchbac, srchbre, srchcha, srchgol

References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

Purpose

Sum squared error performance function

Syntax

```
perf = sse(E,X,PP)
perf = sse(E,net,PP)
info = sse(code)
```

Description

`sse` is a network performance function. It measures performance according to the sum of squared errors.

`sse(E,X,PP)` takes from one to three arguments,

- `E` - Matrix or cell array of error vector(s).
- `X` - Vector of all weight and bias values (ignored).
- `PP` - Performance parameters (ignored).

and returns the sum squared error.

`sse(E,net,PP)` can take an alternate argument to `X`,

- `net` - Neural network from which `X` can be obtained (ignored).

`sse(code)` returns useful information for each `code` string:

- '`deriv`' - Name of derivative function.
- '`name`' - Full name.
- '`pnames`' - Names of training parameters.
- '`pdefaults`' - Default training parameters.

Examples

Here a two-layer feed-forward is created with a 1-element input ranging from -10 to 10, four hidden `tansig` neurons, and one `purelin` output neuron.

```
net = newff([-10 10],[4 1],{'tansig','purelin'});
```

Here the network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the sum squared error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
perf = sse(e)
```

Note that sse can be called with only one argument because the other arguments are ignored. sse supports those arguments to conform to the standard performance function argument list.

Network Use

To prepare a custom network to be trained with sse, set net.performFcn to 'sse'. This will automatically set net.performParam to the empty matrix [], as sse has no performance parameters.

Calling train or adapt will result in sse being used to calculate performance.

See Also

dsse

Purpose

Sum squared elements of a matrix

Syntax

```
sumsqr(m)
```

Description

`sumsqr(M)` returns the sum of the squared elements in `M`.

Examples

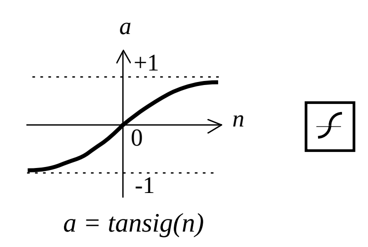
```
s = sumsqr([1 2;3 4])
```

tansig

Purpose

Hyperbolic tangent sigmoid transfer function

Graph and Symbol



a = tansig(n)

Tan-Sigmoid Transfer Function

Syntax

```
A = tansig(N)
info = tansig(code)
```

Description

`tansig` is a transfer function. Transfer functions calculate a layer's output from its net input.

`tansig(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns each element of `N` squashed between -1 and 1.

`tansig(code)` return useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

`tansig` is named after the hyperbolic tangent, which has the same shape. However, `tanh` may be more accurate and is recommended for applications that require the hyperbolic tangent.

Examples

Here is the code to create a plot of the `tansig` transfer function.

```
n = -5:0.1:5;
a = tansig(n);
plot(n,a)
```

Network Use

You can create a standard network that uses `tansig` by calling `newff` or `newcf`.

To change a network so a layer uses `tansig`, set `net.layers{i,j}.transferFcn` to 'tansig'.

In either case, call `sim` to simulate the network with `tansig`. See `newff` or `newcf` for simulation examples.

Algorithm

`tansig(N)` calculates its output according to:

```
n = 2/(1+exp(-2*n))-1
```

This is mathematically equivalent to `tanh(N)`. It differs in that it runs faster than the MATLAB implementation of `tanh`, but the results can have very small numerical differences. This function is a good trade off for neural networks, where speed is important and the exact shape of the transfer function is not.

See Also

`sim, dtansig, logsig`

References

Vogl, T. P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, “Accelerating the convergence of the backpropagation method,” *Biological Cybernetics*, vol. 59, pp. 257-263, 1988.

train

Purpose Train a neural network

Syntax

```
[net,tr,Y,E,Pf,Af] = train(net,P,T,Pi,Ai,VV,TV)
```

To Get Help Type `help network/train`

Description

`train` trains a network `net` according to `net.trainFcn` and `net.trainParam`.

`train(NET,P,T,Pi,Ai,VV,TV)` takes,

- `net` - Neural Network.
- `P` - Network inputs.
- `T` - Network targets, default = zeros.
- `Pi` - Initial input delay conditions, default = zeros.
- `Ai` - Initial layer delay conditions, default = zeros.
- `VV` - Structure of validation vectors, default = [].
- `TV` - Structure of test vectors, default = [].

and returns,

- `net` - New network.
- `TR` - Training record (`epoch` and `perf`).
- `Y` - Network outputs
- `E` - Network errors.
- `Pf` - Final input delay conditions.
- `Af` - Final layer delay conditions.

Note that `T` is optional and need only be used for networks that require targets. `Pi` and `Pf` are also optional and need only be used for networks that have input or layer delays.

Optional arguments `VV` and `TV` are described below.

`train`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

P - Ni x TS cell array, each element P{i,ts} is an Ri x Q matrix.

T - Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix.

Pi - Ni x ID cell array, each element Pi{i,k} is an Ri x Q matrix.

Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

Y - NO x TS cell array, each element Y{i,ts} is an Ui x Q matrix.

E - Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix.

Pf - Ni x ID cell array, each element Pf{i,k} is an Ri x Q matrix.

Af - Nl x LD cell array, each element Af{i,k} is an Si x Q matrix.

where

Ni = net.numInputs

Nl = net.numLayers

Nt = net.numTargets

ID = net.numInputDelays

LD = net.numLayerDelays

TS = Number of time steps

Q = Batch size

Ri = net.inputs{i}.size

Si = net.layers{i}.size

Vi = net.targets{i}.size

The columns of Pi, Pf, Ai, and Af are ordered from the oldest delay condition to the most recent:

Pi{i,k} = input i at time ts=k-ID.

Pf{i,k} = input i at time ts=TS+k-ID.

Ai{i,k} = layer output i at time ts=k-LD.

Af{i,k} = layer output i at time ts=TS+k-LD.

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument into a single matrix:

- `P  - (sum of Ri)` x `Q` matrix
- `T  - (sum of Vi)` x `Q` matrix
- `Pi - (sum of Ri)` x `(ID*Q)` matrix.
- `Ai - (sum of Si)` x `(LD*Q)` matrix.
- `Y  - (sum of Ui)` x `Q` matrix.
- `E  - (sum of Vi)` x Q matrix
- `Pf - (sum of Ri)` x `(ID*Q)` matrix.
- `Af - (sum of Si)` x `(LD*Q)` matrix.

If `VV` and `TV` are supplied they should be an empty matrix [] or a structure with the following fields:

- `VV.P,  TV.P` - Validation/test inputs.
- `VV.T,  TV.T` - Validation/test targets, default = zeros.
- `VV.Pi, TV.Pi` - Validation/test initial input delay conditions, default = zeros.
- `VV.Ai, TV.Ai` - Validation/test layer delay conditions, default = zeros.

The validation vectors are used to stop training early if further training on the primary vectors will hurt generalization to the validation vectors. Test vector performance can be used to measure how well the network generalizes beyond primary and validation vectors. If `VV.T`, `VV.Pi`, or `VV.Ai` are set to an empty matrix or cell array, default values will be used. The same is true for `TV.T`, `TV.Pi`, `TV.Ai`.

Examples

Here input `P` and targets `T` define a simple function which we can plot:

```
p = [0 1 2 3 4 5 6 7 8];
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28 0.66 0.99];
plot(p,t,'o')
```

Here `newff` is used to create a two-layer feed-forward network. The network will have an input (ranging from 0 to 8), followed by a layer of 10 `tansig` neurons, followed by a layer with 1 `purelin` neuron. `trainlm` backpropagation is used. The network is also simulated.

```
net = newff([0 8],[10 1],{'tansig' 'purelin'},'trainlm');
```

```
y1 = sim(net,p)
plot(p,t,'o',p,y1,'x')
```

Here the network is trained for up to 50 epochs to a error goal of 0.01, and then resimulated.

```
net.trainParam.epochs = 50;
net.trainParam.goal = 0.01;
net = train(net,p,t);
y2 = sim(net,p)
plot(p,t,'o',p,y1,'x',p,y2,'*')
```

Algorithm

`train` calls the function indicated by `net.trainFcn`, using the training parameter values indicated by `net.trainParam`.

Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function `net.trainFcn` occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly each epoch from concurrent input vectors (or sequences). `newc` and `newsom` return networks that use `trainr`, a training function that presents each input vector once in random order.

See Also

`sim`, `init`, `adapt`, `revert`

trainb

Purpose

Batch training with weight and bias learning rules.

Syntax

```
[net,TR,Ac,El] = trainb(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainb(code)
```

Description

`trainb` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to '`trainb`'.

`trainb` trains a network with weight and bias learning rules with batch updates. The weights and biases are updated at the end of an entire pass through the input data.

`trainb(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed inputs.
- `Tl` - Layer targets.
- `Ai` - Initial input conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Empty matrix [] or structure of validation vectors.
- `TV` - Empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `trainb`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No x Ni x TS` cell array, each element `Pd{i,j,ts}` is a `Dij x Q` matrix.

`Tl - Nl x TS` cell array, each element `P{i,ts}` is a `Vi x Q` matrix or [].

`Ai - Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` or `TV` is not [], it must be a structure of vectors:

`VV.PD, TV.PD` - Validation/test delayed inputs.

`VV.Tl, TV.Tl` - Validation/test layer targets.

`VV.Ai, TV.Ai` - Validation/test initial input conditions.

`VV.Q, TV.Q` - Validation/test batch size.

`VV.TS, TV.TS` - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`trainb(CODE)` returns useful information for each `CODE` string:

`'pnames'` - Names of training parameters.

`'pdefaults'` - Default training parameters.

Network Use

You can create a standard network that uses `trainb` by calling `newlin`.

To prepare a custom network to be trained with `trainb`:

1. Set `net.trainFcn` to '`trainb`'.
   (This will set `NET.trainParam` to `trainb`'s default parameters.)
2. Set each `NET.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `NET.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `NET.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network:

1. Set `NET.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `newlin` for training examples

Algorithm

Each weight and bias updates according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.
- Validation performance has increase more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newp, newlin, train`

trainbfg

Purpose

BFGS quasi-Newton backpropagation

Syntax

```
[net,TR,Ac,El] = trainbfg(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainbfg(code)
```

Description

`trainbfg` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

`trainbfg(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to `trainbfg`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn` `'srchcha'`		Name of line search routine to use.

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor, which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor, which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size.
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size.
`net.trainParam.maxstep`	`100`	Maximum step length.
`net.trainParam.minstep`	`1.0e-6`	Minimum step length.
`net.trainParam.bmax`	`26`	Maximum step size.

Dimensions for these variables are:

- `Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.
- `Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.
- `Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

- `VV.PD` - Validation delayed inputs.
- `VV.Tl` - Validation layer targets.
- `VV.Ai` - Validation initial input conditions.
- `VV.Q` - Validation batch size.
- `VV.TS` - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD - Validation delayed inputs.

TV.Tl - Validation layer targets.

TV.Ai - Validation initial input conditions.

TV.Q - Validation batch size.

TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

trainbfg(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Examples

Here is a problem consisting of inputs P and targets T that we would like to solve with a network.

```
P = [0 1 2 3 4 5];
T = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainbfg network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainbfg');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples

Network Use

You can create a standard network that uses trainbfg with newff, newcf, or newelm.

To prepare a custom network to be trained with `trainbfg`:

1 Set `net.trainFcn` to '`trainbfg`'. This will set `net.trainParam` to `trainbfg`'s default parameters.
2 Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainbfg`.

Algorithm

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is an approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (see reference below) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `trainrp`, `traincgf`, `traincgb`, `trainscg`, `traincgp`, `trainoss`.

References

Gill, P. E.,W. Murray, and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

Purpose

Bayesian regularization backpropagation

Syntax

```
[net,TR,Ac,El] = trainbr(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainbr(code)
```

Description

`trainbr` is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well. The process is called Bayesian regularization.

`trainbr(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
  - `TR.mu` - Adaptive `mu` value.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `trainlm`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.mu`	`0.005`	Marquardt adjustment parameter
`net.trainParam.mu_dec`	`0.1`	Decrease factor for `mu`
`net.trainParam.mu_inc`	`10`	Increase factor for `mu`
`net.trainParam.mu_max`	`1e-10`	Maximum value for `mu`
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.mem_reduc`	`1`	

Factor to use for memory/speed trade-off

`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

`VV.PD` - Validation delayed inputs.

`VV.Tl` - Validation layer targets.

`VV.Ai` - Validation initial input conditions.

`VV.Q` - Validation batch size.

`VV.TS` - Validation time steps.

which is normally used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row.

If `TV` is not [], it must be a structure of validation vectors,

`TV.PD` - Validation delayed inputs.

`TV.Tl` - Validation layer targets.

`TV.Ai` - Validation initial input conditions.

`TV.Q` - Validation batch size.

`TV.TS` - Validation time steps.

which is used to test the generalization capability of the trained network.

`trainbr(code)` returns useful information for each `code` string:

'`pnames`' - Names of training parameters.

'`pdefaults`' - Default training parameters.

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network. It involves fitting a noisy sine wave.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Here a two-layer feed-forward network is created. The network's input ranges from [-1 to 1]. The first layer has 20 `tansig` neurons, the second layer has one `purelin` neuron. The `trainbr` network training function is to be used. The plot of the resulting network output should show a smooth response, without overfitting.

Create a Network

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'trainbr');
```

Train and Test the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net = train(net,p,t);
a = sim(net,p)
plot(p,a,p,t,'+')
```

Network Use

You can create a standard network that uses `trainbr` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainbr`:

1. Set `net.trainFcn` to '`trainlm`'. This will set `net.trainParam` to `trainbr`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainbr`.

See `newff`, `newcf`, and `newelm` for examples.

Algorithm

`trainbr` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities. See MacKay (*Neural Computation*) and Foresee and Hagan (*Proceedings of the International Joint Conference on Neural Networks*) for more detailed discussions of Bayesian regularization.

This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian `jX` of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value `mu` is increased by `mu_inc` until the change shown above results in a reduced performance value. The change is then made to the network and `mu` is decreased by `mu_dec`.

The parameter `mem_reduc` indicates how to use memory and speed to calculate the Jacobian `jX`. If `mem_reduc` is 1, then `trainlm` runs the fastest, but can require a lot of memory. Increasing `mem_reduc` to 2 cuts some of the memory required by a factor of two, but slows `trainlm` somewhat. Higher values continue to decrease the amount of memory needed and increase the training times.

Training stops when any one of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- `mu` exceeds `mu_max`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `trainrp`, `traincgf`, `traincgb`, `trainscg`, `traincgp`, `trainoss`

References

Foresee, F. D., and M. T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997.

MacKay, D. J. C., "Bayesian interpolation," *Neural Computation*, vol. 4, no. 3, pp. 415-447, 1992.

trainc

Purpose

Cyclical order incremental training with learning functions

Syntax

```
[net,TR,Ac,El] = trainc(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainc(code)
```

Description

`trainc` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainc'`.

`trainc` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in cyclic order.

`trainc(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed inputs.
- `Tl` - Layer targets.
- `Ai` - Initial input conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Ignored.
- `TV` - Ignored.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
- `Ac` - Collective layer outputs.
- `El` - Layer errors.

Training occurs according to the `trainc`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No x Ni x TS` cell array, each element `Pd{i,j,ts}` is a `Dij x Q` matrix.

`Tl - Nl x TS` cell array, each element `P{i,ts}` is a `Vi x Q` matrix or [].

`Ai - Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

`trainc` does not implement validation or test vectors, so arguments `VV` and `TV` are ignored.

`trainc(code)` returns useful information for each `code` string:

`'pnames'`	- Names of training parameters.
`'pdefaults'`	- Default training parameters.

Network Use

You can create a standard network that uses `trainc` by calling `newp`.

To prepare a custom network to be trained with `trainc`:

1. Set `net.trainFcn` to '`trainc`'.
   (This will set `net.trainParam` to `trainc` default parameters.)
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}.learnFcn` to a learning function.

4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network:

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `newp` for training examples.

Algorithm

For each epoch, each vector (or sequence) is presented in order to the network with the weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.

See Also

`newp`, `newlin`, `train`

Purpose Conjugate gradient backpropagation with Powell-Beale restarts

Syntax

```
[net,TR,Ac,El] = traincgb(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgb(code)
```

Description

`traincgb` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

`traincgb(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.
- `TV` - Either empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the traincgb's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.show	25	Epochs between showing progress
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to train in seconds
net.trainParam.min_grad	1e-6	Minimum performance gradient
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.searchFcn 'srchcha'		Name of line search routine to use.

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol 20

Divide into delta to determine tolerance for linear search.

net.trainParam.alpha 0.001

Scale factor, which determines sufficient reduction in perf.

net.trainParam.beta 0.1

Scale factor, which determines sufficiently large step size.

net.trainParam.delta 0.01

Initial step size in interval location step.

net.trainParam.gama 0.1

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in srch_cha.)

net.trainParam.low_lim	0.1	Lower limit on change in step size.
net.trainParam.up_lim	0.5	Upper limit on change in step size.
net.trainParam.maxstep	100	Maximum step length.
net.trainParam.minstep	1.0e-6	Minimum step length.
net.trainParam.bmax	26	Maximum step size.

Dimensions for these variables are:

- Pd - No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.
- Tl - Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.
- Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

- VV.PD - Validation delayed inputs.
- VV.Tl - Validation layer targets.
- VV.Ai - Validation initial input conditions.
- VV.Q - Validation batch size.
- VV.TS - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

- TV.PD - Validation delayed inputs.
- TV.Tl - Validation layer targets.
- TV.Ai - Validation initial input conditions.
- TV.Q - Validation batch size.
- TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

traincgb(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgb network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgb');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

Network Use

You can create a standard network that uses traincgb with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgb:

1. Set net.trainFcn to 'traincgb'. This will set net.trainParam to traincgb's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traincgb.

Algorithm

`traincgb` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula:

```
dX = -gX + dX_old*Z;
```

where `gX` is the gradient. The parameter `Z` can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming,* for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `traincgp`, `traincgf`, `traincgb`, `trainscg`, `trainoss`, `trainbfg`

References

Powell, M. J. D.,“Restart procedures for the conjugate gradient method,” *Mathematical Programming*, vol. 12, pp. 241-254, 1977.

traincgf

Purpose Conjugate gradient backpropagation with Fletcher-Reeves updates

Syntax

```
[net,TR,Ac,El] = traincgf(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgf(code)
```

Description

`traincgf` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Fletcher-Reeves updates.

`traincgf(NET,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `traincgf`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn` `'srchcha'`		Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor, which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor, which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size.
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size.
`net.trainParam.maxstep`	`100`	Maximum step length.
`net.trainParam.minstep`	`1.0e-6`	Minimum step length.
`net.trainParam.bmax`	`26`	Maximum step size.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

`VV.PD` - Validation delayed inputs.

`VV.Tl` - Validation layer targets.

`VV.Ai` - Validation initial input conditions.

`VV.Q` - Validation batch size.

`VV.TS` - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row.

If `TV` is not `[]`, it must be a structure of validation vectors,

`TV.PD` - Validation delayed inputs.

`TV.Tl` - Validation layer targets.

`TV.Ai` - Validation initial input conditions.

`TV.Q` - Validation batch size.

`TV.TS` - Validation time steps.

which is used to test the generalization capability of the trained network.

`traincgf(code)` returns useful information for each `code` string:

'`pnames`' - Names of training parameters.

'`pdefaults`' - Default training parameters.

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

Network Use

You can create a standard network that uses `traincgf` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`:

1. Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `traincgf`.

Algorithm

`traincgf` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction, according to the formula:

```
dX = -gX + dX_old*Z;
```

where `gX` is the gradient. The parameter `Z` can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

```
Z=normnew_sqr/norm_sqr;
```

where `norm_sqr` is the norm square of the previous gradient and `normnew_sqr` is the norm square of the current gradient. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `traincgp`, `traincgb`, `trainscg`, `traincgp`, `trainoss`, `trainbfg`

References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

traincgp

Purpose

Conjugate gradient backpropagation with Polak-Ribiere updates

Syntax

```
[net,TR,Ac,El] = traincgp(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgp(code)
```

Description

`traincgp` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Polak-Ribiere updates.

`traincgp(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `traincgp`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn` `'srchcha'`		Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size.
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size.
`net.trainParam.maxstep`	`100`	Maximum step length.
`net.trainParam.minstep`	`1.0e-6`	Minimum step length.
`net.trainParam.bmax`	`26`	Maximum step size.

Dimensions for these variables are:

```
Pd - No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.
Tl - Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.
Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.
```

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD - Validation delayed inputs.
VV.Tl - Validation layer targets.
VV.Ai - Validation initial input conditions.
VV.Q - Validation batch size.
VV.TS - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD - Validation delayed inputs.
TV.Tl - Validation layer targets.
TV.Ai - Validation initial input conditions.
TV.Q - Validation batch size.
TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

traincgp(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgp network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgp');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

Network Use

You can create a standard network that uses traincgp with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgp:

1. Set net.trainFcn to 'traincgp'. This will set net.trainParam to traincgp's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traincgp.

Algorithm

traincgp can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula:

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. For the Polak-Ribiere variation of conjugate gradient it is computed according to:

```
Z = ((gX - gX_old)'*gX)/norm_sqr;
```

where norm_sqr is the norm square of the previous gradient and gX_old is the gradient on the previous iteration. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `trainrp`, `traincgf`, `traincgb`, `trainscg`, `trainoss`, `trainbfg`

References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

traingd

Purpose Gradient descent backpropagation

Syntax

```
[net,TR,Ac,El] = traingd(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingd(code)
```

Description

`traingd` is a network training function that updates weight and bias values according to gradient descent.

`traingd(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either an empty matrix `[]` or a structure of validation vectors.
- `TV` - Empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `traingd`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`10`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl - Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix.

`Ai - Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` or `TV` is not `[]`, it must be a structure of validation vectors,

`VV.PD, TV.PD` - Validation/test delayed inputs.

`VV.Tl, TV.Tl` - Validation/test layer targets.

`VV.Ai, TV.Ai` - Validation/test initial input conditions.

`VV.Q, TV.Q` - Validation/test batch size.

`VV.TS, TV.TS` - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingd

`traingd(code)` returns useful information for each `code` string:

'`pnames`' - Names of training parameters.

'`pdefaults`' - Default training parameters.

Network Use

You can create a standard network that uses `traingd` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traingd`:

1. Set `net.trainFcn` to '`traingd`'. This will set `net.trainParam` to `traingd`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `traingd`.

See `newff`, `newcf`, and `newelm` for examples.

Algorithm

`traingd` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent:

```
dX = lr * dperf/dX
```

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff, newcf, traingdm, traingda, traingdx, trainlm`

Purpose

Gradient descent with adaptive learning rate backpropagation

Syntax

```
[net,TR,Ac,El] = traingda(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingda(code)
```

Description

`traingda` is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate.

`traingda(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
  - `TR.lr` - Adaptive learning rate.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the traingda's training parameters, shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.max_perf_inc	1.04	Maximum performance increase
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between showing progress
net.trainParam.time	inf	Maximum time to train in seconds

Dimensions for these variables are:

Pd - No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl - Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of validation vectors,

VV.PD, TV.PD - Validation/test delayed inputs.

VV.Tl, TV.Tl - Validation/test layer targets.

VV.Ai, TV.Ai - Validation/test initial input conditions.

VV.Q, TV.Q - Validation/test batch size.

VV.TS, TV.TS - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingda(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Network Use

You can create a standard network that uses traingda with newff, newcf, or newelm.

To prepare a custom network to be trained with traingda:

1 Set net.trainFcn to 'traingda'. This will set net.trainParam to traingda's default parameters.
2 Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traingda.

See newff, newcf, and newelm for examples.

Algorithm

traingda can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance dperf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

```
dX = lr*dperf/dX
```

At each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.

- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingd`, `traingdm`, `traingdx`, `trainlm`

Purpose

Gradient descent with momentum backpropagation

Syntax

```
[net,TR,Ac,El] = traingdm(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingdm(code)
```

Description

`traingdm` is a network training function that updates weight and bias values according to gradient descent with momentum.

`traingdm(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the traingdm's training parameters shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.mc	0.9	Momentum constant.
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between showing progress
net.trainParam.time	inf	Maximum time to train in seconds

Dimensions for these variables are:

Pd - No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl - Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of validation vectors,

VV.PD, TV.PD - Validation/test delayed inputs.

VV.Tl, TV.Tl - Validation/test layer targets.

VV.Ai, TV.Ai - Validation/test initial input conditions.

VV.Q, TV.Q - Validation/test batch size.

VV.TS, TV.TS - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail

epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingdm(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Network Use

You can create a standard network that uses traingdm with newff, newcf, or newelm.

To prepare a custom network to be trained with traingdm:

1. Set net.trainFcn to 'traingdm'. This will set net.trainParam to traingdm's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traingdm.

See newff, newcf, and newelm for examples.

Algorithm

traingdm can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

where dXprev is the previous change to the weight or bias.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increase more than max_fail times since the last time it decreased (when using validation).

See Also

newff, newcf, traingd, traingda, traingdx, trainlm

traingdx

Purpose

Gradient descent with momentum and adaptive learning rate backpropagation

Syntax

```
[net,TR,Ac,El] = traingdx(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingdx(code)
```

Description

`traingdx` is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

`traingdx(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
  - `TR.lr` - Adaptive learning rate.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `traingdx`'s training parameters shown here with their default values:

Parameter	Default	Description
`net.trainParam.epochs`	`10`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.lr_inc`	`1.05`	Ratio to increase learning rate
`net.trainParam.lr_dec`	`0.7`	Ratio to decrease learning rate
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.max_perf_inc`	`1.04`	Maximum performance increase
`net.trainParam.mc`	`0.9`	Momentum constant.
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl - Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai - Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` or `TV` is not `[]`, it must be a structure of validation vectors,

`VV.PD, TV.PD` - Validation/test delayed inputs.

`VV.Tl, TV.Tl` - Validation/test layer targets.

`VV.Ai, TV.Ai` - Validation/test initial input conditions.

`VV.Q,  TV.Q` - Validation/test batch size.

`VV.TS, TV.TS` - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`traingdx(code)` return useful information for each `code` string:

'`pnames`' - Names of training parameters.

'`pdefaults`' - Default training parameters.

Network Use

You can create a standard network that uses `traingdx` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traingdx`:

1. Set `net.trainFcn` to '`traingdx`'. This will set `net.trainParam` to `traingdx`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `traingdx`.

See `newff`, `newcf`, and `newelm` for examples.

Algorithm

`traingdx` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*mc*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increase more than max_fail times since the last time it decreased (when using validation).

See Also

newff, newcf, traingd, traingdm, traingda, trainlm

Purpose Levenberg-Marquardt backpropagation

Syntax

```
[net,TR] = trainlm(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainlm(code)
```

Description

`trainlm` is a network training function that updates weight and bias values according to Levenberg-Marquardt optimization.

`trainlm(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix [] or structure of validation vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
  - `TR.mu` - Adaptive `mu` value.

Training occurs according to the `trainlm`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.mem_reduc`	`1`	Factor to use for memory/speed tradeoff
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.mu`	`0.001`	Initial Mu
`net.trainParam.mu_dec`	`0.1`	Mu decrease factor
`net.trainParam.mu_inc`	`10`	Mu increase factor
`net.trainParam.mu_max`	`1e10`	Maximum Mu
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl - Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai - Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of vectors,

VV.PD, TV.PD - Validation/test delayed inputs.

VV.Tl, TV.Tl - Validation/test layer targets.

VV.Ai, TV.Ai - Validation/test initial input conditions.

VV.Q, TV.Q - Validation/test batch size.

VV.TS, TV.TS - Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

trainlm(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Network Use

You can create a standard network that uses trainlm with newff, newcf, or newelm.

To prepare a custom network to be trained with trainlm:

1. Set net.trainFcn to 'trainlm'. This will set net.trainParam to trainlm's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with trainlm.

See newff, newcf, and newelm for examples.

Algorithm

trainlm can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate the Jacobian jX of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value `mu` is increased by `mu_inc` until the change above results in a reduced performance value. The change is then made to the network and `mu` is decreased by `mu_dec`.

The parameter `mem_reduc` indicates how to use memory and speed to calculate the Jacobian `jX`. If `mem_reduc` is 1, then `trainlm` runs the fastest, but can require a lot of memory. Increasing `mem_reduc` to 2 cuts some of the memory required by a factor of two, but slows `trainlm` somewhat. Higher values continue to decrease the amount of memory needed and increase training times.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- `mu` exceeds `mu_max`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingd`, `traingdm`, `traingda`, `traingdx`

Purpose One step secant backpropagation

Syntax

```
[net,TR,Ac,El] = trainoss(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainoss(code)
```

Description

`trainoss` is a network training function that updates weight and bias values according to the one step secant method.

`trainoss(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.
- `TV` - Either empty matrix [] or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `trainoss`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn` `'srchcha'`		Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor, which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor, which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size.
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size.
`net.trainParam.maxstep`	`100`	Maximum step length.
`net.trainParam.minstep`	`1.0e-6`	Minimum step length.
`net.trainParam.bmax`	`26`	Maximum step size.

Dimensions for these variables are:

Pd - No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl - Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai - Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD - Validation delayed inputs.

VV.Tl - Validation layer targets.

VV.Ai - Validation initial input conditions.

VV.Q - Validation batch size.

VV.TS - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD - Validation delayed inputs.

TV.Tl - Validation layer targets.

TV.Ai - Validation initial input conditions.

TV.Q - Validation batch size.

TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

`trainoss(code)` returns useful information for each `code` string:

`'pnames'` - Names of training parameters.

`'pdefaults'` - Default training parameters.

Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `trainoss` network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainoss');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

Network Use

You can create a standard network that uses `trainoss` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainoss`:

1. Set `net.trainFcn` to '`trainoss`'. This will set `net.trainParam` to `trainoss`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainoss`.

Algorithm

trainoss can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients according to the following formula:

```
dX = -gX + Ac*X_step + Bc*dgX;
```

where gX is the gradient, X_step is the change in the weights on the previous iteration, and dgX is the change in the gradient from the last iteration. See Battiti (*Neural Computation*) for a more detailed discussion of the one step secant algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, traincgp, trainbfg

References

Battiti, R. “First and second order methods for learning: Between steepest descent and Newton’s method,” *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

trainr

Purpose

Random order incremental training with learning functions.

Syntax

```
[net,TR,Ac,El] = trainr(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainr(code)
```

Description

`trainr` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainr'`.

`trainr` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

`trainr(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed inputs.
- `Tl` - Layer targets.
- `Ai` - Initial input conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Ignored.
- `TV` - Ignored.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
- `Ac` - Collective layer outputs.
- `El` - Layer errors.

Training occurs according to `trainr`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Dimensions for these variables are:

`Pd - No x Ni x TS` cell array, each element `Pd{i,j,ts}` is a `Dij x Q` matrix.

`Tl - Nl x TS` cell array, each element `P{i,ts}` is a `Vi x Q` matrix or [].

`Ai - Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

`trainr` does not implement validation or test vectors, so arguments `VV` and `TV` are ignored.

`trainr(code)` returns useful information for each `code` string:

`'pnames'` - Names of training parameters.

`'pdefaults'` - Default training parameters.

Network Use

You can create a standard network that uses `trainr` by calling `newc` or `newsom`.

To prepare a custom network to be trained with `trainr`:

1 Set `net.trainFcn` to `'trainr'`.
   (This will set `net.trainParam` to `trainr`'s default parameters.)

2 Set each `net.inputWeights{i,j}.learnFcn` to a learning function.

3. Set each `net.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network:

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `newc` and `newsom` for training examples.

Algorithm

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.

See Also

`newp`, `newlin`, `train`

Purpose Resilient backpropagation

Syntax

```
[net,TR,Ac,El] = trainrp(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainrp(code)
```

Description

`trainrp` is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (RPROP).

`trainrp(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `trainrp`'s training parameters shown here with their default values:

`net.trainParam.epochs`	100	Maximum number of epochs to train
`net.trainParam.show`	25	Epochs between showing progress
`net.trainParam.goal`	0	Performance goal
`net.trainParam.time`	inf	Maximum time to train in seconds
`net.trainParam.min_grad`	1e-6	Minimum performance gradient
`net.trainParam.max_fail`	5	Maximum validation failures
`net.trainParam.lr`	0.01	Learning rate
`net.trainParam.delt_inc`	1.2	Increment to weight change
`net.trainParam.delt_dec`	0.5	Decrement to weight change
`net.trainParam.delta0`	0.07	Initial weight change
`net.trainParam.deltamax`	50.0	Maximum weight change

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

- VV.PD - Validation delayed inputs.
- VV.Tl - Validation layer targets.
- VV.Ai - Validation initial input conditions.
- VV.Q - Validation batch size.
- VV.TS - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

- TV.PD - Validation delayed inputs.
- TV.Tl - Validation layer targets.
- TV.Ai - Validation initial input conditions.
- TV.Q - Validation batch size.
- TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

trainrp(code) returns useful information for each code string:

- 'pnames' - Names of training parameters.
- 'pdefaults' - Default training parameters.

Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainrp network training function is to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainrp');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

Network Use

You can create a standard network that uses `trainrp` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainrp`:

1. Set `net.trainFcn` to '`trainrp`'. This will set `net.trainParam` to `trainrp`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainrp`.

Algorithm

`trainrp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
dX = deltaX.*sign(gX);
```

where the elements of `deltaX` are all initialized to `delta0` and `gX` is the gradient. At each iteration the elements of `deltaX` are modified. If an element of `gX` changes sign from one iteration to the next, then the corresponding element of `deltaX` is decreased by `delta_dec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltaX` is increased by `delta_inc`. See Reidmiller and Braun, *Proceedings of the IEEE International Conference on Neural Networks*.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.

- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `traincgp`, `traincgf`, `traincgb`, `trainscg`, `trainoss`, `trainbfg`

References

Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco,1993.

trains

Purpose Sequential order incremental training w/learning functions

Syntax

```
[net,TR,Ac,El] = trains(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trains(code)
```

Description

`trains` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trains'`.

`trains` trains a network with weight and bias learning rules with sequential updates. The sequence of inputs is presented to the network with updates occurring after each time step.

This incremental training algorithm is commonly used for adaptive applications.

`trains` takes these inputs:

`net` - Neural network.
`Pd` - Delayed inputs.
`Tl` - Layer targets.
`Ai` - Initial input conditions.
`Q` - Batch size.
`TS` - Time steps.
`VV` - Ignored.
`TV` - Ignored.

and after training the network with its weight and bias learning functions returns:

`net` - Updated network.
`TR` - Training record.
  `TR.time steps` - Number of time steps.
  `TR.perf` - performance for each time step.
`Ac` - Collective layer outputs.
`El` - Layer errors.

Training occurs according to `trains`'s training parameter shown here with its default value:

`net.trainParam.passes` `1` Number of times to present sequence

Dimensions for these variables are:

`Pd - No x NixTS` cell array, each element `P{i,j,ts}` is a `Zij x Q` matrix.

`Tl - Nl x TS` cell array, each element `P{i,ts}` is an `Vi x Q` matrix or [].

`Ai - Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix.

`Ac - Nl x (LD+TS)` cell array, each element `Ac{i,k}` is an `Si x Q` matrix.

`El - Nl x TS` cell array, each element `El{i,k}` is an `Si x Q` matrix or [].

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Zij = Ri * length(net.inputWeights{i,j}.delays)
```

`trains(code)` returns useful information for each `code` string:

`'pnames'` - Names of training parameters.

`'pdefaults'` - Default training parameters.

Network Use

You can create a standard network that uses `trains` for adapting by calling `newp` or `newlin`.

To prepare a custom network to adapt with `trains`:

1. Set `net.adaptFcn` to `'trains'`.
   (This will set `net.adaptParam` to `trains`'s default parameters.)
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}`.learnFcn to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To allow the network to adapt:

1. Set weight and bias learning parameters to desired values.
2. Call `adapt`.

See `newp` and `newlin` for adaption examples.

Algorithm

Each weight and bias is updated according to its learning function after each time step in the input sequence.

See Also

`newp`, `newlin`, `train`, `trainb`, `trainc`, `trainr`

Purpose Scaled conjugate gradient backpropagation

Syntax

```
[net,TR,Ac,El] = trainscg(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainscg(code)
```

Description

`trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

`trainscg(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` - Neural network.
- `Pd` - Delayed input vectors.
- `Tl` - Layer target vectors.
- `Ai` - Initial input delay conditions.
- `Q` - Batch size.
- `TS` - Time steps.
- `VV` - Either empty matrix `[]` or structure of validation vectors.
- `TV` - Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` - Trained network.
- `TR` - Training record of various values over each epoch:
  - `TR.epoch` - Epoch number.
  - `TR.perf` - Training performance.
  - `TR.vperf` - Validation performance.
  - `TR.tperf` - Test performance.
- `Ac` - Collective layer outputs for last epoch.
- `El` - Layer errors for last epoch.

Training occurs according to the `trainscg`'s training parameters shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.sigma`	`5.0e-5`	Determines change in weight for second derivative approximation.
`net.trainParam.lambda`	`5.0e-7`	Parameter for regulating the indefiniteness of the Hessian.

Dimensions for these variables are:

`Pd` - `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` - `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` - `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

`VV.PD` - Validation delayed inputs.

`VV.Tl` - Validation layer targets.

`VV.Ai` - Validation initial input conditions.

`VV.Q` - Validation batch size.

`VV.TS` - Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD - Validation delayed inputs.

TV.Tl - Validation layer targets.

TV.Ai - Validation initial input conditions.

TV.Q - Validation batch size.

TV.TS - Validation time steps.

which is used to test the generalization capability of the trained network.

trainscg(code) returns useful information for each code string:

'pnames' - Names of training parameters.

'pdefaults' - Default training parameters.

Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainscg network training function is used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainscg');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

Network Use

You can create a standard network that uses `trainscg` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainscg`:

1. Set `net.trainFcn` to '`trainscg`'. This will set `net.trainParam` to `trainscg`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainscg`.

Algorithm

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`.

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincgf` and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainbfg, traincgp, trainoss`

References

Moller, M. F., “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, vol. 6, pp. 525-533, 1993.

tramnmx

Purpose

Transform data using a precalculated minimum and maximum value

Syntax

```
[PN] = tramnmx(P,minp,maxp)
```

Description

`tramnmx` transforms the network input set using minimum and maximum values that were previously computed by `premnmx`. This function needs to be used when a network has been trained using data normalized by `premnmx`. All subsequent inputs to the network need to be transformed using the same normalization.

`tramnmx(P,minp, maxp)`takes these inputs

- `P` - `R` x `Q` matrix of input (column) vectors.
- `minp`- `R` x 1 vector containing original minimums for each input.
- `maxp`- `R` x 1 vector containing original maximums for each input.

and returns,

- `PN` - `R` x `Q` matrix of normalized input vectors

Examples

Here is the code to normalize a given data set, so that the inputs and targets will fall in the range `[-1,1]`, using `premnmx`, and also code to train a network with the normalized data.

```
p = [-10 -7.5 -5 -2.5 0 2.5 5 7.5 10];
t = [0 7.07 -10 -7.07 0 7.07 10 7.07 0];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
```

If we then receive new inputs to apply to the trained network, we will use `tramnmx` to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using `postmnmx`.

```
p2 = [4 -7];
[p2n] = tramnmx(p2,minp,maxp);
an = sim(net,pn);
[a] = postmnmx(an,mint,maxt);
```

Algorithm

```
pn = 2*(p-minp)/(maxp-minp) - 1;
```

See Also

`premnmx, prestd, prepca, trastd, trapca`

trapca

Purpose Principal component transformation

Syntax

```
[Ptrans] = trapca(P,transMat)
```

Description

`trapca` preprocesses the network input training set by applying the principal component transformation that was previously computed by `prepca`. This function needs to be used when a network has been trained using data normalized by `prepca`. All subsequent inputs to the network need to be transformed using the same normalization.

`trapca(P,transMat)` takes these inputs,

- `P` - `R` x `Q` matrix of centered input (column) vectors.
- `transMat` - Transformation matrix.

and returns,

- `Ptrans` - Transformed data set.

Examples

Here is the code to perform a principal component analysis and retain only those components that contribute more than two percent to the variance in the data set. `prestd` is called first to create zero mean data, which is needed for `prepca`.

```
p = [-1.5 -0.58 0.21 -0.96 -0.79; -2.2 -0.87 0.31 -1.4  -1.2];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.02);
net = newff(minmax(ptrans),[5 1],{'tansig''purelin'},'trainlm');
net = train(net,ptrans,tn);
```

If we then receive new inputs to apply to the trained network, we will use `trastd` and `trapca` to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using `poststd`.

```
p2 = [1.5 -0.8;0.05 -0.3];
[p2n] = trastd(p2,meanp,stdp);
[p2trans] = trapca(p2n,transMat)
an = sim(net,p2trans);
[a] = poststd(an,meant,stdt);
```

Algorithm

```
Ptrans = transMat*P;
```

See Also

prestd, premnmx, prepca, trastd, tramnmx

trastd

Purpose

Preprocess data using a precalculated mean and standard deviation

Syntax

```
[PN] = trastd(P,meanp,stdp)
```

Description

`trastd` preprocesses the network training set using the mean and standard deviation that were previously computed by `prestd`. This function needs to be used when a network has been trained using data normalized by `prestd`. All subsequent inputs to the network need to be transformed using the same normalization.

`trastd(P,T)` takes these inputs,

- `P` - `R` x `Q` matrix of input (column) vectors.
- `meanp` - `R` x 1 vector containing the original means for each input.
- `stdp` - `R` x 1 vector containing the original standard deviations for each input.

and returns,

- `PN` - `R` x `Q` matrix of normalized input vectors.

Examples

Here is the code to normalize a given data set so that the inputs and targets will have means of zero and standard deviations of 1.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
```

If we then receive new inputs to apply to the trained network, we will use `trastd` to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using `poststd`.

```
p2 = [1.5 -0.8;0.05 -0.3];
[p2n] = trastd(p2,meanp,stdp);
an = sim(net,pn);
[a] = poststd(an,meant,stdt);
```

Algorithm

```
pn = (p-meanp)/stdp;
```

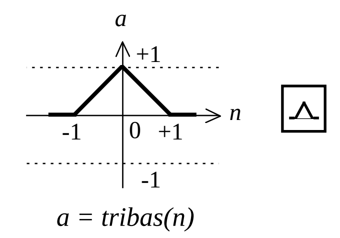
See Also

`premnmx, prepca, prestd, trapca, tramnmx`

Purpose

Triangular basis transfer function

Graph and Symbol



Triangular Basis Function

Syntax

```
A = tribas(N)
info = tribas(code)
```

Description

`tribas` is a transfer function. Transfer functions calculate a layer's output from its net input.

`tribas(N)` takes one input,

`N` - `S` x `Q` matrix of net input (column) vectors.

and returns each element of `N` passed through a radial basis function.

`tribas(code)` returns useful information for each `code` string:

'`deriv`' - Name of derivative function.

'`name`' - Full name.

'`output`' - Output range.

'`active`' - Active input range.

Examples

Here we create a plot of the `tribas` transfer function.

```
n = -5:0.1:5;
a = tribas(n);
plot(n,a)
```

Network Use

To change a network so that a layer uses `tribas`, set `net.layers{i}.transferFcn` to '`tribas`'.

Call `sim` to simulate the network with `tribas`.

Algorithm

`tribas(N)` calculates its output with according to:

`tribas(n) = 1-abs(n)`, if -1 <= n <= 1; = 0, otherwise.

See Also

`sim`, `radbas`

Purpose

Convert vectors to indices

Syntax

```
ind = vec2ind(vec)
```

Description

`ind2vec` and `vec2ind` allow indices to be represented either by themselves or as vectors containing a 1 in the row of the index they represent.

`vec2ind(vec)` takes one argument,

`vec` - Matrix of vectors, each containing a single 1.

and returns the indices of the 1's.

Examples

Here four vectors (each containing only one "1" element) are defined and the indices of the 1's are found.

```
vec = [1 0 0 0; 0 0 1 0; 0 1 0 1]
ind = vec2ind(vec)
```

See Also

`ind2vec`

A

Glossary

ADALINE - An acronym for a linear neuron: ADAptive LINear Element.

adaption - A training method that proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented.

adaptive learning rate - A learning rate that is adjusted according to an algorithm during training to minimize training time.

adaptive filter - A network that contains delays and whose weights are adjusted after each new input vector is presented. The network "adapts" to changes in the input signal properties if such occur. This kind of filter is used in long distance telephone lines to cancel echoes.

architecture - A description of the number of the layers in a neural network, each layer's transfer function, the number of neurons per layer, and the connections between layers.

backpropagation learning rule - A learning rule in which weights and biases are adjusted by error-derivative (delta) vectors backpropagated through the network. Backpropagation is commonly applied to feedforward multilayer networks. Sometimes this rule is called the generalized delta rule.

backtracking search - Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained.

batch - A matrix of input (or target) vectors applied to the network "simultaneously." Changes to the network weights and biases are made just once for the entire set of vectors in the input matrix. (This term is being replaced by the more descriptive expression "concurrent vectors.")

batching - The process of presenting a set of input vectors for simultaneous calculation of a matrix of output vectors and/or new weights and biases.

Bayesian framework - Assumes that the weights and biases of the network are random variables with specified distributions.

BFGS quasi-Newton algorithm - A variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.

bias - A neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.

bias vector - A column vector of bias values for a layer of neurons.

Brent's search - A linear search that is a hybrid combination of the golden section search and a quadratic interpolation.

Charalambous' search - A hybrid line search that uses a cubic interpolation, together with a type of sectioning.

cascade forward network - A layered network in which each layer only receives inputs from previous layers.

classification - An association of an input vector with a particular target vector.

competitive layer - A layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector.

competitive learning - The unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.

competitive transfer function - Accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the "winner," the neuron associated with the most positive element of the net input **n**.

concurrent input vectors - Name given to a matrix of input vectors that are to be presented to a network "simultaneously." All the vectors in the matrix will be used in making just one set of changes in the weights and biases.

conjugate gradient algorithm - In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.

connection - A one-way link between neurons in a network.

connection strength - The strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.

cycle - A single presentation of an input vector, calculation of output, and new weights and biases.

dead neurons - A competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.

decision boundary - A line, determined by the weight and bias vectors, for which the net input n is zero.

delta rule - See the Widrow-Hoff learning rule.

delta vector - The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.

distance - The distance between neurons, calculated from their positions with a distance function.

distance function - A particular way of calculating distance, such as the Euclidean distance between two vectors.

early stopping - A technique based on dividing the data into three subsets. The first subset is the training set used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design.

epoch - The presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

error jumping - A sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

error ratio - A training parameter used with adaptive learning rate and momentum training of backpropagation networks.

error vector - The difference between a network's output vector in response to an input vector and an associated target output vector.

feedback network - A network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.

feedforward network - A layered network in which each layer only receives inputs from previous layers.

Fletcher-Reeves update - A method developed by Fletcher and Reeves for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

function approximation - The task performed by a network trained to respond to inputs with an approximation of a desired function.

generalization - An attribute of a network whose output for a new input vector tends to be close to outputs for similar input vectors in its training set.

generalized regression network - Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.

global minimum - The lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.

golden section search - A linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.

gradient descent - The process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

hard-limit transfer function - A transfer that maps inputs greater-than or equal-to 0 to 1, and all other values to 0.

Hebb learning rule - Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and post-weight neurons.

hidden layer - A layer of a network that is not connected to the network output. (For instance, the first layer of a two-layer feedforward network.)

home neuron - A neuron at the center of a neighborhood.

hybrid bisection-cubicsearch - A line search that combines bisection and cubic interpolation.

input layer - A layer of neurons receiving inputs directly from outside the network.

initialization - The process of setting the network weights and biases to their original values.

input space - The range of all possible input vectors.

input vector - A vector presented to the network.

input weights - The weights connecting network inputs to layers.

input weight vector - The row vector of weights going to a neuron.

Jacobian matrix - Contains the first derivatives of the network errors with respect to the weights and biases.

Kohonen learning rule - A learning rule that trains selected neuron's weight vectors to take on the values of the current input vector.

layer - A group of neurons having connections to the same inputs and sending outputs to the same destinations.

layer diagram - A network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias and weight matrices are shown. Individual neurons and connections are not shown. (See Chapter 2.)

layer weights - The weights connecting layers to other layers. Such weights need to have non-zero delays if they form a recurrent connection (i.e., a loop).

learning - The process by which weights and biases are adjusted to achieve some desired network behavior.

learning rate - A training parameter that controls the size of weight and bias changes during learning.

learning rules - Methods of deriving the next changes that might be made in a network OR a procedure for modifying the weights and biases of a network.

Levenberg-Marquardt - An algorithm that trains a neural network 10 to 100 faster than the usual gradient descent backpropagation method. It will always compute the approximate Hessian matrix, which has dimensions n-by-n.

line search function - Procedure for searching along a given search direction (line) to locate the minimum of the network performance.

linear transfer function - A transfer function that produces its input as its output.

link distance - The number of links, or steps, that must be taken to get to the neuron under consideration.

local minimum - The minimum of a function over a limited range of input values. A local minimum may not be the global minimum.

log-sigmoid transfer function - A squashing function of the form shown below that maps the input to the interval (0,1). (The toolbox function is `logsig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

Manhattan distance - The Manhattan distance between two vectors **x** and **y** is calculated as:

```
D = sum(abs(x-y))
```

maximum performance increase - The maximum amount by which the performance is allowed to increase in one iteration of the variable learning rate training algorithm.

maximum step size - The maximum step size allowed during a linear search. The magnitude of the weight vector is not allowed to increase by more than this maximum step size in one iteration of a training algorithm.

mean square error function - The performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.

momentum - A technique often used to make it less likely for a backpropagation networks to get caught in a shallow minima.

momentum constant - A training parameter that controls how much "momentum" is used.

mu parameter - The initial value for the scalar μ.

neighborhood - A group of neurons within a specified distance of a particular neuron. The neighborhood is specified by the indices for all of the neurons that lie within a radius d of the winning neuron i^*:

$$N_i(d) = \{j, d_{ij} \leq d\}$$

net input vector - The combination, in a layer, of all the layer's weighted input vectors with its bias.

neuron - The basic processing element of a neural network. Includes weights and bias, a summing junction and an output transfer function. Artificial

neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

neuron diagram - A network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.

ordering phase - Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

output layer - A layer whose output is passed to the world outside the network.

output vector - The output of a neural network. Each element of the output vector is the output of a neuron.

output weight vector - The column vector of weights coming from a neuron or input. (See outstar learning rule.)

outstar learning rule - A learning rule that trains a neuron's (or input's) output weight vector to take on the values of the current output vector of the post-weight layer. Changes in the weights are proportional to the neuron's output.

overfitting - A case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.

pass - Each traverse through all of the training input and target vectors.

pattern - A vector.

pattern association - The task performed by a network trained to respond with the correct output vector for each presented input vector.

pattern recognition - The task performed by a network trained to respond when an input vector close to a learned vector is presented. The network "recognizes" the input as one of the original target vectors.

performance function - Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type `nnets` and look under performance functions.

perceptron - A single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.

perceptron learning rule - A learning rule for training single-layer hard-limit networks. It is guaranteed to result in a perfectly functioning network in finite time, given that the network is capable of doing so.

performance - The behavior of a network.

Polak-Ribiére update - A method developed by Polak and Ribiére for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

positive linear transfer function - A transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

postprocessing - Converts normalized outputs back into the same units that were used for the original targets.

Powell-Beale restarts - A method developed by Powell and Beale for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient.

preprocessing - Perform some transformation of the input or target data before it is presented to the neural network.

principal component analysis - Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.

quasi-Newton algorithm - Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.

radial basis networks - A neural network that can be designed directly by fitting special response elements where they will do the most good.

radial basis transfer function - The transfer function for a radial basis neuron is:

$$radbas(n) = e^{-n^2}$$

regularization - Involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights.

resilient backpropagation - A training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid “squashing” transfer functions.

saturating linear transfer function - A function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is `satlin`.)

scaled conjugate gradient algorithm - Avoids the time consuming line search of the standard conjugate gradient algorithm.

sequential input vectors - A set of vectors that are to be presented to a network “one after the other.” The network weights and biases are adjusted on the presentation of each input vector.

sigma parameter - Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.

sigmoid - Monotonic S-shaped function mapping numbers in the interval $(-\infty,\infty)$ to a finite interval such as (-1,+1) or (0,1).

simulation - Takes the network input **p**, and the network object `net`, and returns the network outputs **a.**

spread constant - The distance an input vector must be from a neuron’s weight vector to produce an output of 0.5.

squashing function - A monotonic increasing function that takes input values between $-\infty$ and $+\infty$ and returns values in a finite interval.

star learning rule - A learning rule that trains a neuron’s weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron’s output.

sum-squared error - The sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

supervised learning - A learning process in which changes in a network’s weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.

symmetric hard-limit transfer function - A transfer that maps inputs greater-than or equal-to 0 to +1, and all other values to -1.

symmetric saturating linear transfer function - Produces the input as its output as long as the input i in the range -1 to 1. Outside that range the output is -1 and +1 respectively.

tan-sigmoid transfer function - A squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is `tansig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

tapped delay line - A sequential set of delays with outputs available at each delay output.

target vector - The desired output vector for a given input vector.

test vectors - A set of input vectors (not used directly in training) that is used to test the trained network.

topology functions - Ways to arrange the neurons in a grid, box, hexagonal, or random topology.

training - A procedure whereby a network is adjusted to do a particular job. Commonly viewed as an "offline" job, as opposed to an adjustment made during each time interval as is done in adaptive training.

training vector - An input and/or target vector used to train a network.

transfer function - The function that maps a neuron's (or layer's) net output **n** to its actual output.

tuning phase - Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

underdetermined system - A system that has more variables than constraints.

unsupervised learning - A learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases.

update - Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.

validation vectors - A set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.

weighted input vector - The result of applying a weight to a layer's input, whether it is a network input or the output of another layer.

weight function - Weight functions apply weights to an input to get weighted inputs as specified by a particular function.

weight matrix - A matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix W refers to the connection strength from input j to neuron i.

Widrow-Hoff learning rule - A learning rule used to trained single-layer linear networks. This rule is the predecessor of the backpropagation rule and is sometimes referred to as the delta rule.

B

Bibliography

[Batt92] Battiti, R., “First and second order methods for learning: Between steepest descent and Newton’s method,” *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

[Beal72] Beale, E. M. L., “A derivation of conjugate gradients,” in F. A. Lootsma, ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.

[Bren73] Brent, R. P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

[Caud89] Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[CaBu92] Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations, Vols. 1 and 2,* Cambridge, MA: the MIT Press, 1992.

This is a two volume workbook designed to give students “hands on” experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear and helpful in understanding a field that traditionally has been buried in mathematics.

[Char92] Charalambous, C.,“Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992.

[ChCo91] Chen, S., C. F. N. Cowan, and P. M. Grant, “Orthogonal least squares learning algorithm for radial basis function networks,” *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 302-309, 1991.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[DARP88] *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and

discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

[DeSc83] Dennis, J. E., and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[Elma90] Elman, J. L.,"Finding structure in time," *Cognitive Science*, vol. 14, pp. 179-211, 1990.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

[FlRe64] Fletcher, R., and C. M. Reeves, "Function minimization by conjugate gradients," *Computer Journal*, vol. 7, pp. 149-154, 1964.

[FoHa97] Foresee, F. D., and M. T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, pages 1930-1935, 1997.

[GiMu81] Gill, P. E., W. Murray, and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

[Gros82] Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan,M.T. and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642-1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, 1999, pp. 311-340.

[HaMe94] Hagan, M. T., and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

[HDB96] Hagan, M. T., H. B. Demuth, and M. H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has demonstration programs, an instructor's guide and transparency overheads for teaching.

[Hebb49] Hebb, D. O., *The Organization of Behavior,* New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

[Himm72] Himmelblau, D. M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

[Joll86] Jolliffe, I. T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

[HuSb92] Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System - A Survey," *Automatica*, Vol. 28, 1992, pp. 1083-1112.

[Koho87] Kohonen, T., *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

[Koho97] Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications and hardware of self-organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A. N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405-1422, 1989.

This paper discusses a class of neural networks described by first order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li et. al. is implemented in Chapter 9 of this *User's Guide*.

[Lipp87] Lippman, R. P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, pp. 4-22, 1987.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

[MacK92] MacKay, D. J. C., "Bayesian interpolation," *Neural Computation*, vol. 4, no. 3, pp. 415-447, 1992.

[McPi43] McCulloch, W. S., and W. H. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

[Moll93] Moller, M. F., "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, pp. 525-533, 1993.

[MuNe92]Murray, R., D. Neumerkel, and D. Sbarbaro, "Neural Networks for Modeling and Control of a Non-linear Dynamic System," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404-409.

[NaMu97]Narendra, K.S. and S. Mukhopadhyay, "Adaptive Control Using Neural Networks and Approximate Models," *IEEE Transactions on Neural Networks* Vol. 8, 1997, pp. 475-485.

[NgWi89] Nguyen, D., and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," *Proceedings of the International Joint Conference on Neural Networks*, vol 2, pp. 357-363, 1989.

This paper describes a two-layer network that first learned the truck dynamics and then learned how to back the truck to a specified position at a loading dock. To do this, the neural network had to solve a highly nonlinear control systems problem.

[NgWi90] Nguyen, D., and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *Proceedings of the International Joint Conference on Neural Networks*, vol 3, pp. 21-26, 1990.

Nguyen and Widrow demonstrate that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

[Powe77] Powell, M. J. D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, vol. 12, pp. 241-254, 1977.

[Pulu92] N. Purdie, E.A. Lucas and M.B. Talley, “Direct measure of total cholesterol and its distribution among major serum lipoproteins,” Clinical Chemistry, vol. 38, no. 9, pp. 1645-1647, 1992.

[RiBr93] Riedmiller, M., and H. Braun, “A direct adaptive method for faster backpropagation learning: The RPROP algorithm,” *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[Rose61] Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt’s results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

[RuHi86a] Rumelhart, D. E., G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,”, in D. E. Rumelhart and J. L. McClelland, eds. *Parallel Data Processing, vol.1*, Cambridge, MA: The M.I.T. Press, pp. 318-362, 1986.

This is a basic reference on backpropagation.

[RuHi86b] Rumelhart, D. E., G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.

[RuMc86] Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, eds., *Parallel Distributed Processing, Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of publication.

[Scal85] Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[SoHa96] Soloway, D. and P.J. Haley, “Neural Generalized Predictive Control,” *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277-281.

[VoMa88] Vogl, T. P., J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, “Accelerating the convergence of the backpropagation method,” *Biological Cybernetics*, vol. 59, pp. 256-264, 1988.

Backpropagation learning can be speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

[Wass93] Wasserman, P. D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

[WiHo60] Widrow, B., and M. E. Hoff, “Adaptive switching circuits,” *1960 IRE WESCON Convention Record, New York IRE*, pp. 96-104, 1960.

[WiSt85] Widrow, B., and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

C

Demonstrations and Applications

Tables of Demonstrations and Applications C-2
Chapter 2: Neuron Model and Network Architectures C-2
Chapter 3: Perceptrons C-2
Chapter 4: Linear Filters C-3
Chapter 5: Backpropagation C-3
Chapter 7: Radial Basis Networks C-4
Chapter 8: Self-Organizing and Learn. Vector Quant. Nets . C-4
Chapter 9: Recurrent Networks C-4
Chapter 10: Adaptive Networks C-5
Chapter 11: Applications C-5

Tables of Demonstrations and Applications

Chapter 2: Neuron Model and Network Architectures

	Filename	Page
Simple neuron and transfer functions	nnd2n1	page 2-5
Neuron with vector input	nnd2n2	page 2-7

Chapter 3: Perceptrons

	Filename	Page
Decision boundaries	nnd4db	page 3-5
Perceptron learning rule. Pick boundaries	nnd4pr	page 3-15
Classification with a two-input perceptron	demop1	page 3-20
Outlier input vectors	demop4	page 3-21
Normalized perceptron rule	demop5	page 3-22
Linearly nonseparable vectors	demop6	page 3-21

Chapter 4: Linear Filters

	Filename	Page
Pattern association showing error surface	`demolin1`	page 4-9
Training a linear neuron	`demolin2`	page 4-17
Linear classification system	`nnd10lc`	page 4-17
Linear fit of nonlinear problem	`demolin4`	page 4-18
Underdetermined problem	`demolin5`	page 4-18
Linearly dependent problem	`demolin6`	page 4-19
Too large a learning rate	`demolin7`	page 4-19

Chapter 5: Backpropagation

	Filename	Page
Generalization	`nnd11gn`	page 5-51
Steepest descent backpropagation	`nnd12sd1`	page 5-11
Momentum backpropagation	`nnd12mo`	page 5-13
Variable learning rate backpropagation	`nnd12vl`	page 5-15
Conjugate gradient backpropagation	`nnd12cg`	page 5-19
Marquardt backpropagation	`nnd12m`	page 5-30
Sample training session	`demobp1`	page 5-30

Chapter 7: Radial Basis Networks

	Filename	Page
Radial basis approximation	demorb1	page 7-8
Radial basis underlapping neurons	demorb3	page 7-8
Radial basis overlapping neurons	demorb4	page 7-8
GRNN function approximation	demogrn1	page 7-11
PNN classification	demopnn1	page 7-14

Chapter 8: Self-Organizing and Learn. Vector Quant. Nets

	Filename	Page
Competitive learning	democ1	page 8-8
One-dimensional self-organizing map	demosm1	page 8-23
Two-dimensional self-organizing map	demosm2	page 8-23
Learning vector quantization	demolvq1	page 8-38

Chapter 9: Recurrent Networks

	Filename	Page
Hopfield two neuron design	demohop1	page 9-1
Hopfield unstable equilibria	demohop2	page 9-14
Hopfield three neuron design	demohop3	page 9-14
Hopfield spurious stable points	demohop4	page 9-14

Chapter 10: Adaptive Networks

	Filename	Page
Adaptive noise cancellation, toolbox example	demolin8	page 10-16
Adaptive noise cancellation in airplane cockpit	nnd10nc	page 10-14

Chapter 11: Applications

	Filename	Page
Linear design	applin1	page 11-3
Adaptive linear prediction	applin2	page 11-7
Elman amplitude detection	appelm1	page 11-11
Character recognition	appcr1	page 11-16

D

Simulink

Block Set D-2
Transfer Function Blocks D-2
Net Input Blocks D-3
Weight Blocks D-3
Control Systems Blocks D-4

Block Generation D-5
Example D-5
Exercises D-7

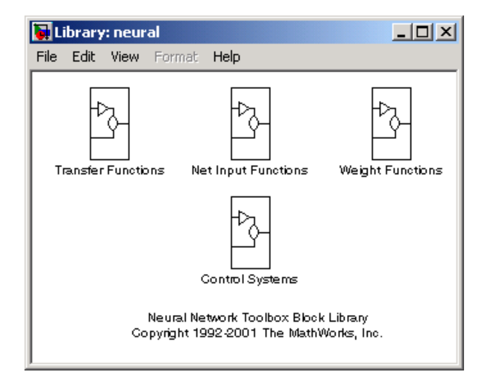
Block Set

The Neural Network Toolbox provides a set of blocks you can use to build neural networks in Simulink or which can be used by the function `gensim` to generate the Simulink version of any network you have created in MATLAB.

Bring up the Neural Network Toolbox blockset with this command.

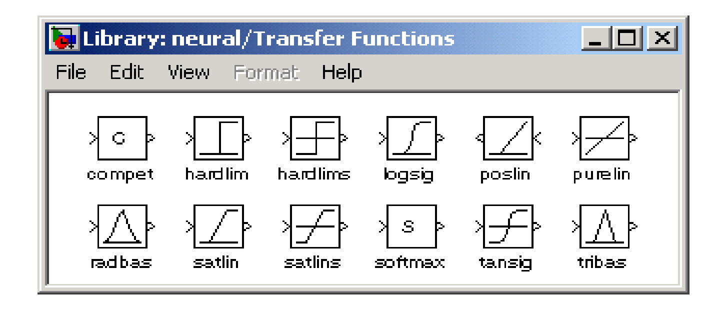
```
neural
```

The result is a window that contains three blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

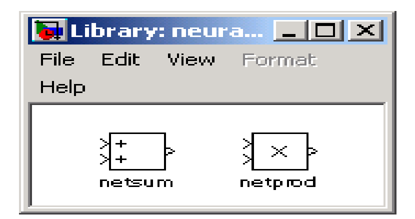
Double-click on the Transfer Functions block in the **Neural** window to bring up a window containing several transfer function blocks.



Each of these blocks takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector.

Net Input Blocks

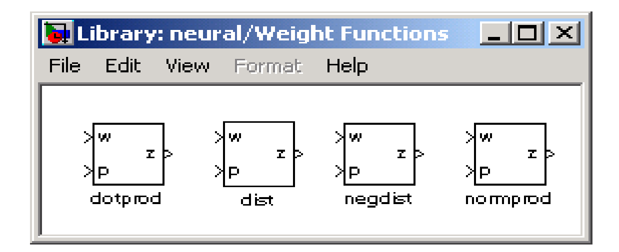
Double-click on the Net Input Functions block in the **Neural** window to bring up a window containing two net-input function blocks.



Each of these blocks takes any number of weighted input vectors, weight layer output vectors, and bias vectors, and returns a net-input vector.

Weight Blocks

Double-click on the Weight Functions block in the **Neural** window to bring up a window containing three weight function blocks.



Each of these blocks takes a neuron's weight vector and applies it to an input vector (or a layer output vector) to get a weighted input value for a neuron.

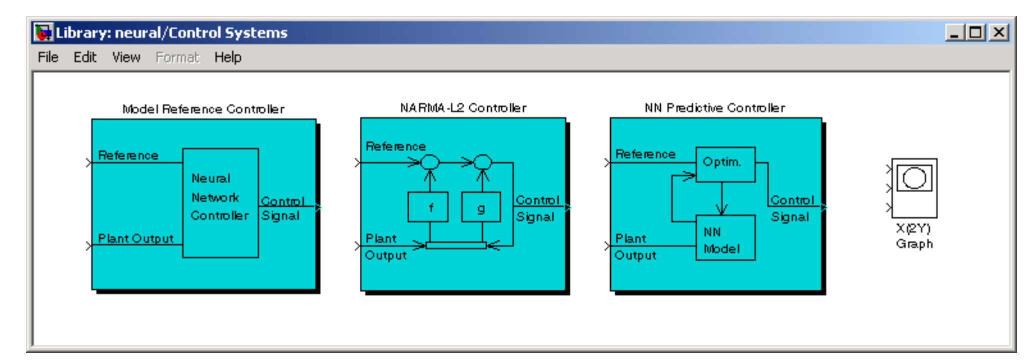
It is important to note that the blocks above expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

It is also important to note that because of this limitation you have to create S weight function blocks (one for each row), to implement a weight matrix going to a layer with S neurons.

This contrasts with the other two kinds of blocks. Only one net input function and one transfer function block are required for each layer.

Control Systems Blocks

Double-click on the Control Systems block in the **Neural** window to bring up a window containing four control systems blocks.



Chapter 6, "Control Systems" describes the application of these control systems blocks.

Block Generation

The function `gensim` generates block descriptions of networks so you can simulate them in Simulink.

```
gensim(net,st)
```

The second argument to `gensim` determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated with its input weights or layer weights, this value can be set to -1. A value of -1 tells `gensim` to generate a network with continuous sampling.

Example

Here is a simple problem defining a set of inputs `p` and corresponding targets `t`.

```
p = [1 2 3 4 5];
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p,t)
```

We can test the network on our original inputs with `sim`.

```
y = sim(net,p)
```

The results returned show the network has solved the problem.

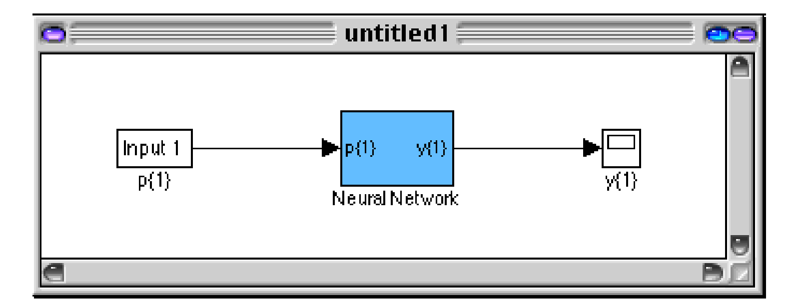
```
y =
       1.0000    3.0000    5.0000    7.0000    9.0000
```

Call `gensim` as follows to generate a Simulink version of the network.

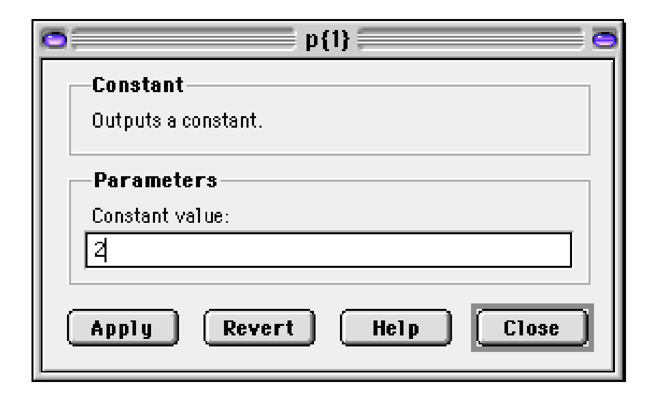
```
gensim(net,-1)
```

The second argument is -1 so the resulting network block samples continuously.

The call to `gensim` results in the following screen. It contains a Simulink system consisting of the linear network connected to a sample input and a scope.



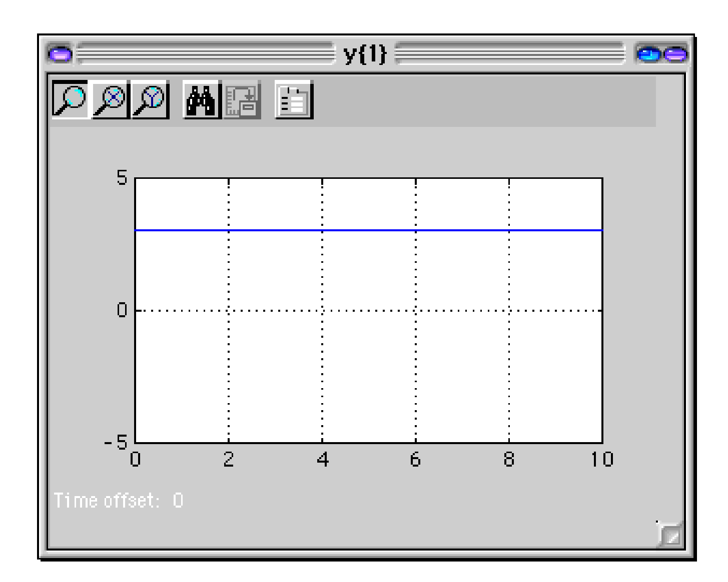
To test the network, double-click on the `Input 1` block at left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to `2`, and then select **Close**.

Select **Start** from the **Simulation** menu. Simulink momentarily pauses as it simulates the system.

When the simulation is over, double-click the scope at the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

Exercises

Here are a couple of exercises you can try.

Changing Input Signal

Replace the constant input block with a signal generator from the standard Simulink block set Sources. Simulate the system and view the network's response.

Discrete Sample Time

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net,0.5)
```

Again replace the constant input with a signal generator. Simulate the system and view the network's response.

E

Code Notes

Dimensions E-2

Variables E-3
Utility Function Variables E-4

Functions E-7

Code Efficiency E-8

Argument Checking E-9

Dimensions

The following code dimensions are used in describing both the network signals that user's commonly see, and those used by the utility functions:

Ni = number of network inputs = net.numInputs

Ri = number of elements in input i = net.inputs{i}.size

Nl = number of layers = net.numLayers

Si = number of neurons in layer i = net.layers{i}.size

Nt = number of targets = net.numTargets

Vi = number of elements in target i, equal to Sj, where j is the ith layer with a target. (A layer n has a target if net.targets(n) == 1.)

No = number of network outputs = net.numOutputs

Ui = number of elements in output i, equal to Sj, where j is the ith layer with an output (A layer n has an output if net.outputs(n) == 1.)

ID = number of input delays = net.numInputDelays

LD = number of layer delays = net.numLayerDelays

TS = number of time steps

Q = number of concurrent vectors or sequences.

Variables

The variables a user commonly uses when defining a simulation or training session are:

P

Network inputs.

Ni-by-TS cell array, each element P{i,ts} is an Ri-by-Q matrix.

Pi

Initial input delay conditions.

Ni-by-ID cell array, each element Pi{i,k} is an Ri-by-Q matrix.

Ai

Initial layer delay conditions.

Nl-by-LD cell array, each element Ai{i,k} is an Si-by-Q matrix.

T

Network targets.

Nt-by-TS cell array, each element P{i,ts} is an Vi-by-Q matrix.

These variables are returned by simulation and training

calls:

Y

Network outputs.

No-by-TS cell array, each element Y{i,ts} is a Ui-by-Q matrix.

E

Network errors.

Nt-by-TS cell array, each element P{i,ts} is an Vi-by-Q matrix.

perf

network performance

Utility Function Variables

These variables are used only by the utility functions.

Pc

Combined inputs.

Ni-by-(ID+TS) cell array, each element P{i,ts} is an Ri-by-Q matrix.

Pc = [Pi P] = Initial input delay conditions and network inputs.

Pd

Delayed inputs.

Ni-by-Nj-by-TS cell array, each element Pd{i,j,ts} is an (Ri*IWD(i,j))-by-Q matrix, where IWD(i,j) is the number of delay taps associated with input weight to layer i from input j.

Equivalently, IWD(i,j) = length(net.inputWeights{i,j}.delays).

Pd is the result of passing the elements of P through each input weights tap delay lines. Since inputs are always transformed by input delays in the same way it saves time to only do that operation once, instead of for every training step.

BZ

Concurrent bias vectors.

Nl-by-1 cell array, each element BZ{i} is a Si-by-Q matrix.

Each matrix is simply Q copies of the net.b{i} bias vector.

IWZ

Weighted inputs.

Ni-by-Nl-by-TS cell array, each element IWZ{i,j,ts} is a Si-by--by-Q matrix.

LWZ

Weighed layer outputs.

Ni-by-Nl-by-TS cell array, each element LWZ{i,j,ts} is a Si-by-Q matrix.

N

Net inputs.

Ni-by-TS cell array, each element N{i,ts} is a Si-by-Q matrix.

A

Layer outputs.

Nl-by-TS cell array, each element A{i,ts} is a Si-by-Q matrix.

Ac

Combined layer outputs.

Nl-by-(LD+TS) cell array, each element A{i,ts} is a Si-by-Q matrix.

Ac = [Ai A] = Initial layer delay conditions and layer outputs.

Tl

Layer targets.

Nl-by-TS cell array, each element Tl{i,ts} is a Si-by-Q matrix.

Tl contains empty matrices [] in rows of layers i not associated with targets, indicated by net.targets(i) == 0.

El

Layer errors.

Nl-by-TS cell array, each element El{i,ts} is a Si-by-Q matrix.

El contains empty matrices [] in rows of layers i not associated with targets, indicated by net.targets(i) == 0.

X

Column vector of all weight and bias values.

Functions

The following functions are the utility functions that you can call to perform a lot of the work of simulating or training a network. You can read about them in their respective help comments.

These functions calculate signals.

`calcpd`, `calca`, `calca1`, `calce`, `calce1`, `calcperf`

These functions calculate derivatives, Jacobians, and values associated with Jacobians.

`calcgx`, `calcjx`, `calcjejj`

`calcgx` is used for gradient algorithms; `calcjx` and `calcjejj` can be used for calculating approximations of the Hessian for algorithms like Levenberg-Marquardt.

These functions allow network weight and bias values to be accessed and altered in terms of a single vector X.

`setx`, `getx`, `formx`

Code Efficiency

The functions sim, train, and adapt all convert a network object to a structure,

```
net = struct(net);
```

before simulation and training, and then recast the structure back to a network.

```
net = class(net,'network')
```

This is done for speed efficiency since structure fields are accessed directly, while object fields are accessed using the MATLAB object method handling system. If users write any code that uses utility functions outside of sim, train, or adapt, they should use the same technique.

Argument Checking

These functions are only recommended for advanced users.

None of the utility functions do any argument checking, which means that the only feedback you get from calling them with incorrectly sized arguments is an error.

The lack of argument checking allows these functions to run as fast as possible.

For “safer” simulation and training, use `sim`, `train` and `adapt`.

Index

A

ADALINE network
  decision boundary 4-5, 10-5
adaption
  custom function 12-30
  function 13-9
  parameters 13-12
adaptive filter 10-9
  example 10-10
  noise cancellation example 10-14
  prediction application 11-7
  prediction example 10-13
  training 2-18
adaptive linear networks 10-2, 10-18
amplitude detection 11-11
applications
  adaptive filtering 10-9
  aerospace 1-5
  automotive 1-5
  banking 1-5
  defense 1-6
  electronics 1-6
  entertainment 1-6
  financial 1-6
  insurance 1-6
  manufacturing 1-6
  medical 1-7, 5-66
  oil and gas exploration 1-7
  robotics 1-7
  speech 1-7
  telecommunications 1-7
  transportation 1-7
architecture
  bias connection 12-5, 13-3
  input connection 12-5, 13-3
architecture (*continued*)
  input delays 13-5
  layer connection 12-5, 13-4
  layer delays 13-6
  number of inputs 12-4, 13-2
  number of layers 12-4, 13-2
  number of outputs 12-6, 13-5
  number of targets 12-6, 13-5
  output connection 12-6, 13-4
  target connection 12-6, 13-4

B

backpropagation 5-2, 6-2
  algorithm 5-9
  example 5-66
backtracking search 5-25
batch training 2-18, 2-20, 5-9
  dynamic networks 2-22
  static networks 2-20
Bayesian framework 5-53
benchmark 5-32, 5-58
BFGS quasi-Newton algorithm 5-26
bias
  connection 12-5
  definition 2-2
  initialization function 13-26
  learning 13-26
  learning function 13-27
  learning parameters 13-27
  subobject 12-10, 13-26
  value 12-11, 13-15
box distance 8-16
Brent's search 5-24

C

cell array
  derivatives 12-34
  errors 12-29
  initial layer delay states 12-28
  input P 2-17
  input vectors 12-13
  inputs and targets 2-20
  inputs property 12-7
  layer targets 12-28
  layers property 12-8
  matrix of concurrent vectors 2-17
  matrix of sequential vectors 2-19
  sequence of outputs 2-15
  sequential inputs 2-15
  tap delayed inputs 12-28
  weight matrices and bias vectors 12-12
Charalambous' search 5-25
classification 7-12
  input vectors 3-4
  linear 4-15
  regions 3-5
code
  mathematical equivalents 2-10
  perceptron network 3-7
  writing 2-5
competitive layer 8-3
competitive neural network 8-4
  example 8-7
competitive transfer function 7-12, 8-3, 8-17
concurrent inputs 2-13, 2-16
conjugate gradient algorithm 5-17
  Fletcher-Reeves update 5-18
  Polak-Ribiere update 5-20
  Powell-Beale restarts 5-21
  scaled 5-22
continuous stirred tank reactor 6-6
control
  control design 6-2
  electromagnet 6-18, 6-19
  feedback linearization 6-2, 6-14
  model predictive control 6-2, 6-3, 6-5, 6-6, 6-8, 6-38
  model reference control 6-2, 6-3, 6-23, 6-25, 6-26, 6-38
  NARMA-L2 6-2, 6-3, 6-14, 6-16, 6-18, 6-38
  plant 6-2, 6-3, 6-23
  robot arm 6-25
  time horizon 6-5
  training data 6-11
CSTR 6-6
custom neural network 12-2

D

dead neurons 8-5
decision boundary 4-5, 10-5
  definition 3-5
demonstrations
  appelm1 11-11
  applin3 11-10
  definition 1-2
  demohop1 9-14
  demohop2 9-14
  demorb4 7-8
  nnd10lc 4-17
  nnd11gn 5-51
  nnd12cg 5-19
  nnd12m 5-30
  nnd12mo 5-13
  nnd12sd1 5-11, 5-24
  nnd12vl 5-15

distance 8-9, 8-14
box 8-16
custom function 12-38
Euclidean 8-14
link 8-16
Manhattan 8-16
tuning phase 8-18
dynamic networks 2-14, 2-16
training 2-19, 2-22

E

early stopping 1-4, 5-55
electromagnet 6-18, 6-19
Elman network 9-3
recurrent connection 9-3
Euclidean distance 8-14
export 6-31
networks 6-31, 6-32
training data 6-35

F

feedback linearization 6-2, 6-14
feedforward network 5-6
finite impulse response filter 4-11, 10-10
Fletcher-Reeves update 5-18

G

generalization 5-51
regularization 5-52
generalized regression network 7-9
golden section search 5-23
gradient descent algorithm 5-2, 5-9
batch 5-10
with momentum 5-11, 5-12
graphical user interface 1-3, 3-23
gridtop topology 8-10

H

Hagan, Martin xxiv, xxv
hard limit transfer function 2-3, 2-25, 3-4
heuristic techniques 5-14
hidden layer
definition 2-12
home neuron 8-15
Hopfield network
architecture 9-8
design equilibrium point 9-10
solution trajectories 9-14
stable equilibrium point 9-10
target equilibrium points 9-10
horizon 6-5
hybrid bisection-cubic search 5-24

I

import 6-31
networks 6-31, 6-32
training data 6-35, 6-36
incremental training 2-18
initial step size function 5-16
initialization
additional functions 12-16
custom function 12-24
definition 3-9
function 13-10
parameters 13-12, 13-13

input
connection 12-5
number 12-4
range 13-17
size 13-17
subobject 12-7, 12-8, 13-17
input vector
outlier 3-21
input vectors
classification 3-4
dimension reduction 5-63
distance 8-9
topology 8-9
input weights
definition 2-10
inputs
concurrent 2-13, 2-16
sequential 2-13, 2-14
installation guide 1-2

J
Jacobian matrix 5-28

K
Kohonen learning rule 8-5

L
lambda parameter 5-22
layer
connection 12-5
dimensions 13-18
distance function 13-18
distances 13-19
initialization function 13-19
layer *(continued)*
net input function 13-20
number 12-4
positions 13-21
size 13-22
subobject 13-18
topology function 13-22
transfer function 13-23
layer weights
definition 2-10
learning rate
adaptive 5-14
maximum stable 4-14
optimal 5-14
ordering phase 8-18
too large 4-19
tuning phase 8-18
learning rules 3-2
custom 12-35
Hebb 12-17
Hebb with decay 12-17
instar 12-17
Kohonen 8-5
outstar 12-17
supervised learning 3-12
unsupervised learning 3-12
Widrow-Hoff 4-13, 5-2, 10-2, 10-4, 10-8, 10-18
learning vector quantization 8-2
creation 8-32
learning rule 8-35, 8-38
LVQ network 8-31
subclasses 8-31
target classes 8-31
union of two subclasses 8-35
least mean square error 4-8, 10-7
Levenberg-Marquardt algorithm 5-28
reduced memory 5-30

line search functions 5-19
backtracking search 5-25
Brent's search 5-24
Charalambous' search 5-25
golden section search 5-23
hybrid bisection-cubic search 5-24
linear networks
design 4-9
linear transfer function 2-3, 2-26, 4-3, 10-3
linear transfer functions 5-5
linearly dependent vectors 4-19
link distance 8-16
log-sigmoid transfer function 2-4, 2-26, 5-4

M

MADALINE 10-4
magnet 6-18, 6-19
Manhattan distance 8-16
maximum performance increase 5-12
maximum step size 5-16
mean square error function 5-9
least 4-8, 10-7
memory reduction 5-30
model predictive control 6-2, 6-3, 6-5, 6-6, 6-8, 6-38
model reference control 6-2, 6-3, 6-23, 6-25, 6-26, 6-38
momentum constant 5-12
mu parameter 5-29

N

NARMA-L2 6-2, 6-3, 6-14, 6-16, 6-18, 6-38
neighborhood 8-9
net input function
custom 12-20
network
definition 12-4
dynamic 2-14, 2-16
static 2-13
network function 12-11
network layer
competitive 8-3
definition 2-6
Network/Data Manager window 3-23
neural network
adaptive linear 10-2, 10-18
competitive 8-4
custom 12-2
definition xviii
feedforward 5-6
generalized regression 7-9
multiple layer 2-11, 5-2, 10-16
one layer 2-8, 3-6, 4-4, 10-4, 10-19
probabilistic 7-12
radial basis 7-2
self organizing 8-2
self-organizing feature map 8-9
Neural Network Design xxiv
Instructor's Manual xxiv
overheads xxiv
neuron
dead (not allocated) 8-5
definition 2-2
home 8-15

Newton's method 5-29
NN predictive control 6-2, 6-3, 6-5, 6-6, 6-8, 6-38
normalization
  inputs and targets 5-61
  mean and standard deviation 5-62
notation
  abbreviated 2-6, 10-17
  layer 2-11
  transfer function symbols 2-4, 2-7
numerical optimization 5-14

O

one step secant algorithm 5-27
ordering phase learning rate 8-18
outlier input vector 3-21
output
  connection 12-6
  number 12-6
  size 13-25
  subobject 12-9, 13-25
output layer
  definition 2-12
  linear 5-6
overdetermined systems 4-18
overfitting 5-51

P

pass
  definition 3-16
pattern recognition 11-16
perceptron learning rule 3-2, 3-13
  normalized 3-22
perceptron network 3-2
  code 3-7
  creation 3-3
  limitations 3-21
performance function 13-10
  custom 12-32
  modified 5-52
  parameters 13-13
plant 6-2, 6-3, 6-23
plant identification 6-9, 6-14, 6-23, 6-27
Polak-Ribiere update 5-20
postprocessing 5-61
post-training analysis 5-64
Powell-Beale restarts 5-21
predictive control 6-2, 6-3, 6-5, 6-6, 6-8, 6-38
preprocessing 5-61
principal component analysis 5-63
probabilistic neural network 7-12
  design 7-13

Q

quasi-Newton algorithm 5-25
  BFGS 5-26

R

radial basis
  design 7-10
  efficient network 7-7
  function 7-2
  network 7-2
  network design 7-5
radial basis transfer function 7-4
recurrent connection 9-3
recurrent networks 9-2

regularization 5-52
automated 5-53
resilient backpropagation 5-16
robot arm 6-25

S

self-organizing feature map (SOFM) network 8-9
neighborhood 8-9
one-dimensional example 8-24
two-dimensional example 8-26
self-organizing networks 8-2
sequential inputs 2-13, 2-14
S-function 14-2
sigma parameter 5-22
simulation 5-8
definition 3-8
Simulink
generating networks D-5
NNT blockset D-2, E-2
spread constant 7-5
squashing functions 5-16
static networks 2-13
batch training 2-20
training 2-18
subobject
bias 12-10, 13-8, 13-26
input 12-7, 12-8, 13-6, 13-17
layer 13-7, 13-18
output 12-9, 13-7, 13-25
target 12-9, 13-7, 13-25
weight 12-10, 13-8, 13-9, 13-28, 13-32
supervised learning 3-12
target output 3-12
training set 3-12
system identification 6-2, 6-4, 6-9, 6-14, 6-23, 6-27

T

tan-sigmoid transfer function 5-5
tapped delay line 4-10, 10-9
target
connection 12-6
number 12-6
size 13-25
subobject 12-9, 13-25
target output 3-12
time horizon 6-5
topologies 8-9
custom function 12-36
gridtop 8-10
topologies for SOFM neuron locations 8-10
training 5-8
batch 2-18, 5-9
competitive networks 8-6
custom function 12-27
definition 2-2, 3-2
efficient 5-61
faster 5-14
function 13-11
incremental 2-18
ordering phase 8-20
parameters 13-14
post-training analysis 5-64
self organizing feature map 8-19
styles 2-18
tuning phase 8-20
training data 6-11
training set 3-12
training styles 2-18
training with noise 11-19

transfer functions
competitive 7-12, 8-3, 8-17
custom 12-18
definition 2-2
derivatives 5-5
hard limit 2-3, 3-4
linear 4-3, 5-5, 10-3
log-sigmoid 2-4, 2-26, 5-4
radial basis 7-4
saturating linear 12-16
soft maximum 12-16
tan-sigmoid 5-5
triangular basis 12-16
transformation matrix 5-63
tuning phase learning rate 8-18
tuning phase neighborhood distance 8-18

U

underdetermined systems 4-18
unsupervised learning 3-12

V

variable learning rate algorithm 5-15
vectors
linearly dependent 4-19

W

weight
definition 2-2
delays 13-28, 13-32
initialization function 13-29, 13-33
learning 13-29, 13-33
learning function 13-30, 13-34
learning parameters 13-31, 13-35
size 13-31, 13-35
subobject 12-10, 13-28, 13-32
value 12-11, 13-14, 13-15
weight function 13-32, 13-36
weight function
custom 12-22
weight matrix
definition 2-8
Widrow-Hoff learning rule 4-13, 5-2, 10-2, 10-4, 10-8, 10-18
workspace (command line) 3-23